

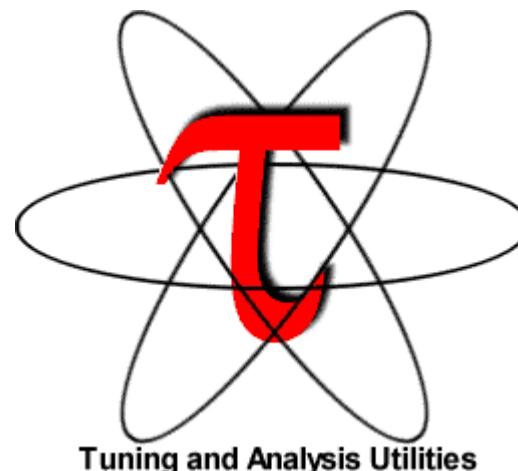
TAU Parallel Performance System

(ACTS Workshop LBL)

Sameer Shende, Allen D. Malony

University of Oregon

{sameer, malony}@cs.uoregon.edu



Tuning and Analysis Utilities



UNIVERSITY
OF OREGON



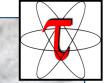
John von Neumann - Institut für Computing
Zentralinstitut für Angewandte Mathematik



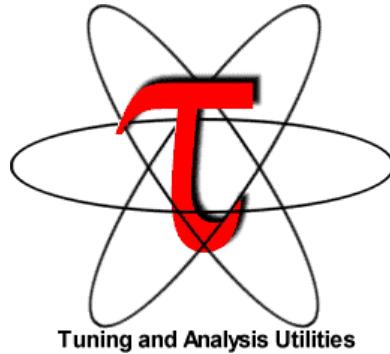


Outline

- Motivation
- Part I: Instrumentation
- Part II: Measurement
- Part III: Analysis Tools
- Conclusion



TAU Performance System Framework



- Tuning and Analysis Utilities
- Performance system framework for scalable parallel and distributed high-performance computing
- Targets a general complex system computation model
 - nodes / contexts / threads
 - Multi-level: system / software / parallelism
 - Measurement and analysis abstraction
- Integrated toolkit for performance instrumentation, measurement, analysis, and visualization
 - Portable, configurable **performance profiling/tracing facility**
 - Open software approach
- University of Oregon, LANL, FZJ Germany
- <http://www.cs.uoregon.edu/research/paracomp/tau>



TAU Performance Systems Goals

- Multi-level performance instrumentation
 - Multi-language automatic source instrumentation
- Flexible and configurable performance measurement
- Widely-ported parallel performance profiling system
 - Computer system architectures and operating systems
 - Different programming languages and compilers
- Support for multiple parallel programming paradigms
 - Multi-threading, message passing, mixed-mode, hybrid
- Support for performance mapping
- Support for object-oriented and generic programming
- Integration in complex software systems and applications



Definitions – Profiling

□ Profiling

- Recording of summary information during execution
 - inclusive, exclusive time, # calls, hardware statistics, ...
- Reflects performance behavior of program entities
 - functions, loops, basic blocks
 - user-defined “semantic” entities
- Very good for low-cost performance assessment
- Helps to expose performance bottlenecks and hotspots
- Implemented through
 - **sampling**: periodic OS interrupts or hardware counter traps
 - **instrumentation**: direct insertion of measurement code



Definitions – Tracing

Tracing

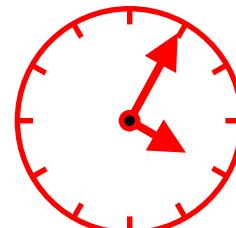
- Recording of information about significant points (**events**) during program execution
 - entering/exiting code region (function, loop, block, ...)
 - thread/process interactions (e.g., send/receive message)
- Save information in **event record**
 - timestamp
 - CPU identifier, thread identifier
 - Event type and event-specific information
- **Event trace** is a time-sequenced stream of event records
- Can be used to reconstruct dynamic program behavior
- Typically requires code instrumentation



Event Tracing: *Instrumentation, Monitor, Trace*

CPU A:

```
void master {  
    trace(ENTER, 1);  
    ...  
    trace(SEND, B);  
    send(B, tag, buf);  
    ...  
    trace(EXIT, 1);  
}
```



timestamp

Event definition

1	master
2	slave
3	...

CPU B:

```
void slave {  
    trace(ENTER, 2);  
    ...  
    recv(A, tag, buf);  
    trace(RECV, A);  
    ...  
    trace(EXIT, 2);  
}
```

MONITOR



...				
58	A	ENTER	1	
60	B	ENTER	2	
62	A	SEND	B	
64	A	EXIT	1	
68	B	RECV	A	
69	B	EXIT	2	
...				

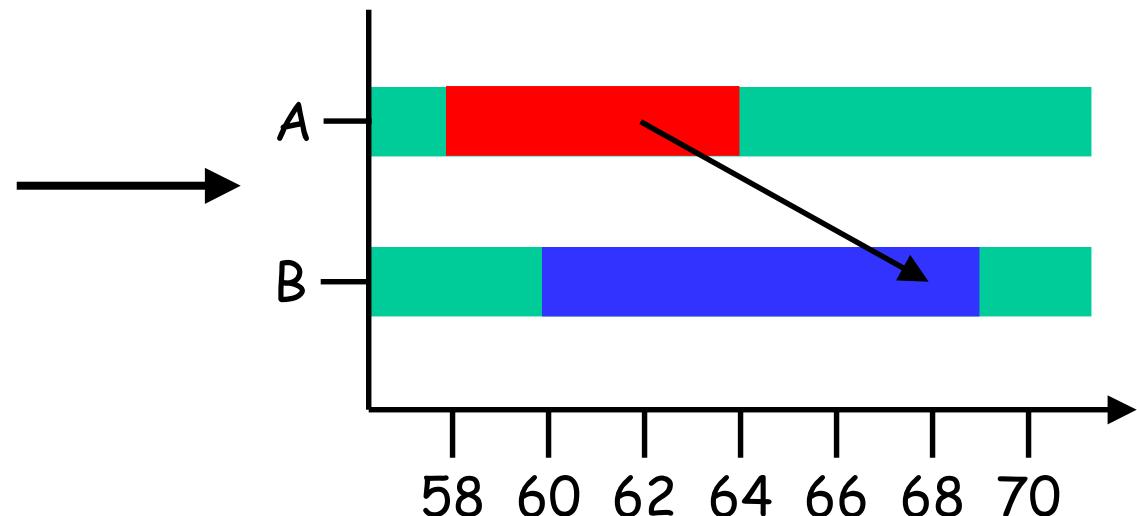


Event Tracing: “Timeline” Visualization

1	master
2	slave
3	...

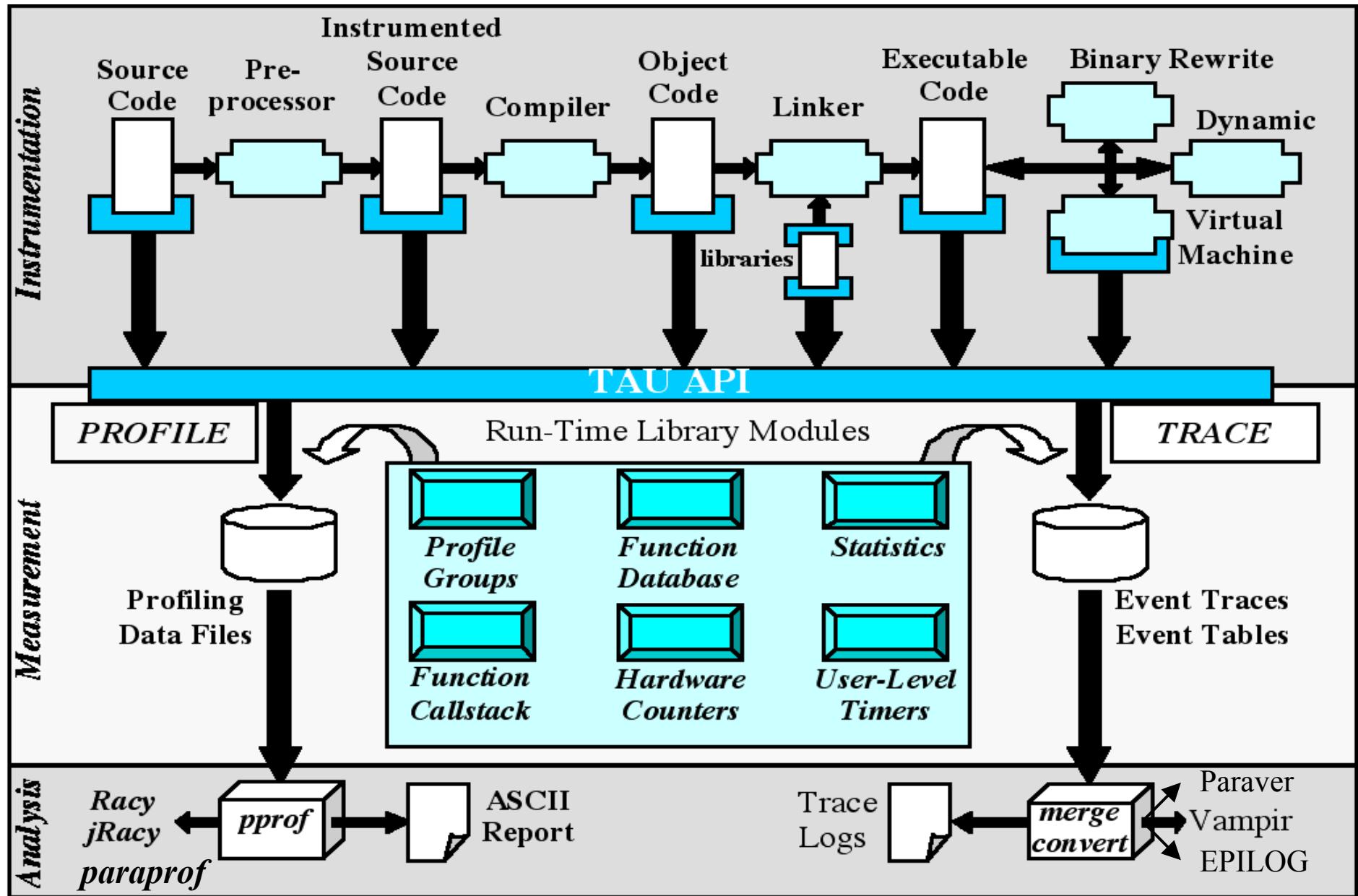
main
master
slave

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			





TAU Performance System Architecture





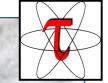
Strategies for Empirical Performance Evaluation

- Empirical performance evaluation as a series of performance experiments
 - Experiment trials describing instrumentation and measurement requirements
 - Where/When/How axes of empirical performance space
 - where are performance measurements made in program
 - routines, loops, statements...
 - when is performance instrumentation done
 - compile-time, while pre-processing, runtime...
 - how are performance measurement/instrumentation options chosen
 - profiling with hw counters, tracing, callpath profiling...



TAU Instrumentation Approach

- Support for standard program events
 - Routines
 - Classes and templates
 - Statement-level blocks
- Support for user-defined events
 - Begin/End events (“user-defined timers”)
 - Atomic events (e.g., size of memory allocated/freed)
 - Selection of event statistics
- Support definition of “semantic” entities for mapping
- Support for event groups
- Instrumentation optimization (eliminate instrumentation in lightweight routines)



TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
 - Source code
 - manual (TAU API, TAU Component API)
 - automatic
 - C, C++, F77/90/95 (Program Database Toolkit (*PDT*))
 - OpenMP (directive rewriting (*Opari*), *POMP spec*)
 - Object code
 - pre-instrumented libraries (e.g., MPI using *PMPI*)
 - statically-linked and dynamically-linked
 - Executable code
 - dynamic instrumentation (pre-execution) (*DynInstAPI*)
 - virtual machine instrumentation (e.g., Java using *JVMPPI*)
 - Proxy Components



Using TAU – A tutorial

- Configuration
 - Manual
 - MPI – Wrapper interposition library
 - PDT- Source rewriting for C,C++, F77/90/95
 - OpenMP – Directive rewriting
 - Component based instrumentation – Proxy components
 - Binary Instrumentation
 - DyninstAPI – Runtime Instrumentation/Rewriting binary
 - Java – Runtime instrumentation
 - Python – Runtime instrumentation
- Measurement
- Performance Analysis



TAU Measurement System Configuration

□ configure [OPTIONS]

- {-c++=<CC>, -cc=<cc>} Specify C++ and C compilers
- {-pthread, -sproc} Use pthread or SGI sproc threads
- -openmp Use OpenMP threads
- -jdk=<dir> Specify Java instrumentation (JDK)
- -opari=<dir> Specify location of Opari OpenMP tool
- -papi=<dir> Specify location of PAPI
- -pdt=<dir> Specify location of PDT
- -dyninst=<dir> Specify location of DynInst Package
- -mpi[inc/lib]=<dir> Specify MPI library instrumentation
- -python[inc/lib]=<dir> Specify Python instrumentation
- -epilog=<dir> Specify location of EPILOG
- -vtf=<dir> Specify location of VTF3 trace package



TAU Measurement System Configuration

- configure [OPTIONS]
 - -TRACE Generate binary TAU traces
 - -PROFILE (default) Generate profiles (summary)
 - -PROFILECALLPATH Generate call path profiles
 - -PROFILEMEMORY Track heap memory for each routine
 - -MULTIPLECOUNTERS Use hardware counters + time
 - -COMPENSATE Compensate timer overhead
 - -CPUTIME Use usertime+system time
 - -PAPIWALLCLOCK Use PAPI's wallclock time
 - -PAPIVIRTUAL Use PAPI's process virtual time
 - -SGITIMERS Use fast IRIX timers
 - -LINUXTIMERS Use fast x86 Linux timers



TAU Measurement Configuration – Examples

- `./configure -c++=xlC_r -pthread`
 - Use TAU with xlC_r and pthread library under AIX
 - Enable TAU profiling (default)
- `./configure -TRACE -PROFILE`
 - Enable both TAU profiling and tracing
- `./configure -c++=xlC_r -cc=xlc_r
-papi=/usr/local/packages/papi
-pdt=/usr/local/pdtoolkit-3.1 -arch=ibm64
-mpiinc=/usr/lpp/ppe.poe/include
-mpilib=/usr/lpp/ppe.poe/lib -MULTIPLECOUNTERS`
 - Use IBM's xlC_r and xlc_r compilers with PAPI, PDT, MPI packages and multiple counters for measurements
- Typically configure multiple measurement libraries



Description of Optional Packages

- **PAPI** – Measures hardware performance data e.g., floating point instructions, L1 data cache misses etc.
- **DyninstAPI** – Helps instrument an application binary at runtime or rewrites the binary
- **EPILOG** – Trace library. Epilog traces can be analyzed by EXPERT [UTK, FZJ], an automated bottleneck detection tool. Part of KOJAK (CUBE, EPILOG, Opari).
- **Opari** – Tool that instruments OpenMP programs
- **Vampir** – Commercial trace visualization tool [Intel]
- **Paraver** – Trace visualization tool [CEPBA]

PAPI Overview



- Performance Application Programming Interface
 - The purpose of the PAPI project is to design, standardize and implement a portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.
- Parallel Tools Consortium project
- University of Tennessee, Knoxville
- <http://icl.cs.utk.edu/papi>





Using TAU

- **Install TAU**

- % configure ; make clean install

- **Instrument application**

- TAU Profiling API

- **Typically modify application makefile**

- include TAU's stub makefile, modify variables

- **Set environment variables**

- directory where profiles/traces are to be stored

- name of merged trace file, retain intermediate trace files, etc.

- **Execute application**

- % mpirun –np <procs> a.out;

- **Analyze performance data**

- paraprof, vampir, pprof, paraver ...



Using TAU – A tutorial

- Configuration
- Instrumentation
 - Manual
 - MPI – Wrapper interposition library
 - PDT- Source rewriting for C,C++, F77/90/95
 - OpenMP – Directive rewriting
 - Component based instrumentation – Proxy components
 - Binary Instrumentation
 - DyninstAPI – Runtime Instrumentation/Rewriting binary
 - Java – Runtime instrumentation
 - Python – Runtime instrumentation
 - Measurement
 - Performance Analysis



TAU Manual Instrumentation API for C/C++

- Initialization and runtime configuration
 - `TAU_PROFILE_INIT(argc, argv);`
`TAU_PROFILE_SET_NODE(myNode);`
`TAU_PROFILE_SET_CONTEXT(myContext);`
`TAU_PROFILE_EXIT(message);`
`TAU_REGISTER_THREAD();`
- Function and class methods for C++ only:
 - `TAU_PROFILE(name, type, group);`
- Template
 - `TAU_TYPE_STRING(variable, type);`
`TAU_PROFILE(name, type, group);`
`CT(variable);`
- User-defined timing
 - `TAU_PROFILE_TIMER(timer, name, type, group);`
`TAU_PROFILE_START(timer);`
`TAU_PROFILE_STOP(timer);`



TAU Measurement API (continued)

- User-defined events
 - TAU_REGISTER_EVENT(variable, event_name);
 - TAU_EVENT(variable, value);
 - TAU_PROFILE_STMT(statement);
- Heap Memory Tracking:
 - TAU_TRACK_MEMORY();
 - TAU_SET_INTERRUPT_INTERVAL(seconds);
 - TAU_DISABLE_TRACKING_MEMORY();
 - TAU_ENABLE_TRACKING_MEMORY();
- Reporting
 - TAU_REPORT_STATISTICS();
 - TAU_REPORT_THREAD_STATISTICS();



Manual Instrumentation – C++ Example

```
#include <TAU.h>

int main(int argc, char **argv)
{
    TAU_PROFILE("int main(int, char **)", " ", TAU_DEFAULT);
    TAU_PROFILE_INIT(argc, argv);
    TAU_PROFILE_SET_NODE(0); /* for sequential programs */
    foo();
    return 0;
}

int foo(void)
{
    TAU_PROFILE("int foo(void)", " ", TAU_DEFAULT); // measures entire foo()
    TAU_PROFILE_TIMER(t, "foo(): for loop", "[23:45 file.cpp]", TAU_USER);
    TAU_PROFILE_START(t);
    for(int i = 0; i < N ; i++){
        work(i);
    }
    TAU_PROFILE_STOP(t);
    // other statements in foo ...
}
```



Manual Instrumentation – F90 Example

```
cc34567 Cubes program - comment line

PROGRAM SUM_OF_CUBES
    integer profiler(2)
    save profiler
    INTEGER :: H, T, U
    call TAU_PROFILE_INIT()
    call TAU_PROFILE_TIMER(profiler, 'PROGRAM SUM_OF_CUBES')
    call TAU_PROFILE_START(profiler)
    call TAU_PROFILE_SET_NODE(0)
! This program prints all 3-digit numbers that
! equal the sum of the cubes of their digits.
    DO H = 1, 9
        DO T = 0, 9
            DO U = 0, 9
                IF (100*H + 10*T + U == H**3 + T**3 + U**3) THEN
                    PRINT "(3I1)", H, T, U
                ENDIF
            END DO
        END DO
    END DO
    call TAU_PROFILE_STOP(profiler)
END PROGRAM SUM_OF_CUBES
```



Compiling

```
% configure [options]  
% make clean install
```

Creates <arch>/lib/Makefile.tau<options> stub Makefile
and <arch>/lib/libTau<options>.a [.so] libraries which defines a single
configuration of TAU



Compiling: TAU Makefiles

- Include TAU Stub Makefile (<arch>/lib) in the user's Makefile.
- Variables:

○ TAU_CXX	Specify the C++ compiler used by TAU
○ TAU_CC, TAU_F90	Specify the C, F90 compilers
○ TAU_DEFS	Defines used by TAU. Add to CFLAGS
○ TAU_LDFLAGS	Linker options. Add to LDFLAGS
○ TAU_INCLUDE	Header files include path. Add to CFLAGS
○ TAU_LIBS	Statically linked TAU library. Add to LIBS
○ TAU_SHLIBS	Dynamically linked TAU library
○ TAU_MPI_LIBS	TAU's MPI wrapper library for C/C++
○ TAU_MPI_FLIBS	TAU's MPI wrapper library for F90
○ TAU_FORTRANLIBS	Must be linked in with C++ linker for F90
○ TAU_CXXLIBS	Must be linked in with F90 linker
○ TAU_INCLUDE_MEMORY	Use TAU's malloc/free wrapper lib
○ TAU_DISABLE	TAU's dummy F90 stub library
○ TAU_COMPILER	Instrument using tau_compiler.sh script

Note: Not including TAU_DEFS in CFLAGS disables instrumentation in C/C++ programs (**TAU_DISABLE** for f90).



Including TAU Makefile - F90 Example

```
include /usr/common/acts/TAU/tau-2.13.7/rs6000/lib/Makefile.tau-pdt
F90 = $(TAU_F90)
FFLAGS = -I<dir>
LIBS = $(TAU_LIBS) $(TAU_CXXLIBS)
OBJS =
TARGET= a.out
TARGET: $(OBJS)
        $(F90) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.f.o:
        $(F90) $(FFLAGS) -c $< -o $@
```



Using MPI Wrapper Interposition Library

Step I: Configure TAU with MPI:

```
% configure -mpiinc=/usr/lpp/ppe.poe/include  
-mpilib=/usr/lpp/ppe.poe/lib -arch=ibm64 -c++=xlc_r  
-cc=xlc_r -pdt=/usr/common/acts/TAU/pdtoolkit-3.2.1  
% make clean; make install
```

Builds <taudir>/<arch>/lib/libTauMpi<options>,
<taudir>/<arch>/lib/Makefile.tau<options> and libTau<options>.a



TAU's MPI Wrapper Interposition Library

- Uses standard MPI Profiling Interface
 - Provides name shifted interface
 - `MPI_Send` = `PMPI_Send`
 - Weak bindings
- Interpose TAU's MPI wrapper library between MPI and TAU
 - `-lmpi` replaced by `-lTauMpi -lpmpi -lmpi`
- No change to the source code! Just **re-link** the application to generate performance data



Including TAU's stub Makefile

```
include /usr/common/acts/TAU/tau-2.13.7/rs6000/lib/Makefile.tau-mpi-pdt

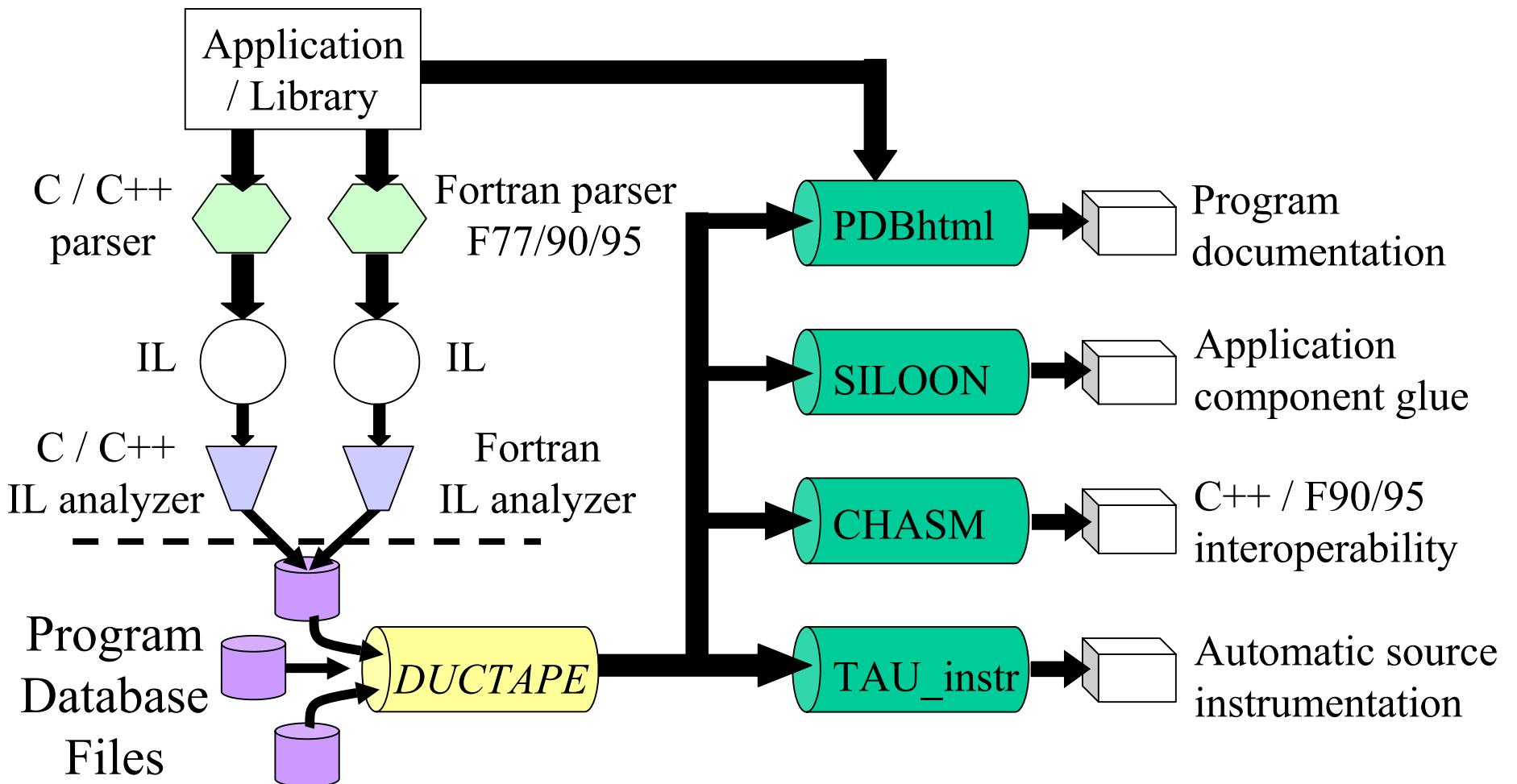
F90 = $(TAU_F90)
CC = $(TAU_CC)
LIBS = $(TAU_MPI_LIBS) $(TAU_LIBS) $(TAU_CXXLIBS)
LD_FLAGS = $(TAU_LDFLAGS)
OBJS =
TARGET= a.out
TARGET: $(OBJS)
        $(CXX) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.f.o:
        $(F90) $(FFLAGS) -c $< -o $@
```



Program Database Toolkit (PDT)

- Program code analysis framework
 - develop source-based tools
- *High-level interface* to source code information
- *Integrated toolkit* for source code parsing, database creation, and database query
 - Commercial grade front-end parsers
 - Portable IL analyzer, database format, and access API
 - Open software approach for tool development
- Multiple source languages
- Implement automatic performance instrumentation tools
 - *tau_instrumentor*

Program Database Toolkit (PDT)





PDT 3.2 Functionality

- C++ statement-level information implementation
 - for, while loops, declarations, initialization, assignment...
 - PDB records defined for most constructs
- DUCTAPE
 - Processes PDB 1.x, 2.x, 3.x uniformly
- PDT applications
 - XMLgen
 - PDB to XML converter
 - Used for CHASM and CCA tools
 - PDBstmt
 - Statement callgraph display tool



PDT 3.2 Functionality (continued)

- Cleanscape Flint parser fully integrated for F90/95
 - Flint parser (f95parse) is very robust
 - Produces PDB records for TAU instrumentation (stage 1)
 - Linux (x86, IA-64, Opteron, Power4), HP Tru64, IBM AIX, Cray X1,T3E, Solaris, SGI, Apple, Windows, Power4 Linux (IBM Blue Gene/L compatible)
 - Full PDB 2.0 specification (stage 2) [SC'04]
 - Statement level support (stage 3) [SC'04]
- URL:
<http://www.cs.uoregon.edu/research/paracomp/pdtoolkit>



Using Program Database Toolkit (PDT)

Step I: Configure PDT:

```
% configure -arch=ibm64 -XLC  
% make clean; make install
```

Builds <pdtdir>/<arch>/bin/cxxparse, cparse, f90parse and f95parse

Builds <pdtdir>/<arch>/lib/libpdb.a. See <pdtdir>/README file.

Step II: Configure TAU with PDT for auto-instrumentation of source code:

```
% configure -arch=ibm64 -c++=xlc -cc=xlc  
-pdt=/usr/contrib/TAU/pdtoolkit-3.1  
% make clean; make install
```

Builds <taudir>/<arch>/bin/tau_instrumentor,

<taudir>/<arch>/lib/Makefile.tau<options> and libTau<options>.a

See <taudir>/INSTALL file.



Using Program Database Toolkit (PDT) (contd.)

1. Parse the Program to create foo.pdb:

```
% cxxparse foo.cpp -I/usr/local/mydir -DMYFLAGS ...
```

or

```
% cparsse foo.c -I/usr/local/mydir -DMYFLAGS ...
```

or

```
% f95parse foo.f90 -I/usr/local/mydir ...
```

2. Instrument the program:

```
% tau_instrumentor foo.pdb    foo.f90 -o foo.inst.f90
```

3. Compile the instrumented program:

```
% ifort foo.inst.f90 -c -I/usr/local/mpi/include -o foo.o
```



TAU Makefile for PDT (C++)

```
include /usr/tau/include/Makefile

CXX = $(TAU_CXX)
CC = $(TAU_CC)

PDTPARSE = $(PDTDIR)=$(PDTARCHDIR)/bin/cxxparse
TAUINSTR = $(TAUROOT)/$(CONFIG_ARCH)/bin/tau_instrumentor
CFLAGS = $(TAU_DEFS) $(TAU_INCLUDE)
LIBS = $(TAU_LIBS)
OBJS = ...
TARGET= a.out
TARGET: $(OBJS)
        $(CXX) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)

.cpp.o:
        $(PDTPARSE) $<
        $(TAUINSTR) $*.pdb $< -o $*.inst.cpp -f select.dat
        $(CC) $(CFLAGS) -c $*.inst.cpp -o $@
```



TAU Makefile for PDT (F90)

```
include $PET_HOME/PTOOLS/tau-2.13.5/rs6000/lib/Makefile.tau-pdt
F90 = $(TAU_F90)
CC = $(TAU_CC)
PDTPARSE = $(PDTDIR)/$(PDTARCHDIR)/bin/f95parse
TAUINSTR = $(TAUROOT)/$(CONFIG_ARCH)/bin/tau_instrumentor
LIBS = $(TAU_LIBS) $(TAU_CXXLIBS)
OBJS =
TARGET= f1.o f2.o f3.o
PDB=merged.pdb
TARGET: $(PDB) $(OBJS)
        $(F90) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
$(PDB): $(OBJS:.o=.f)
        $(PDTF95PARSE) $(OBJS:.o=.f) -o$(PDB) -R free
# This expands to f95parse *.f -o merged.pdb -R free
.f.o:
        $(TAU_INSTR) $(PDB) $< -o $*.inst.f -f sel.dat; \
        $(FCOMPILER) $*.inst.f -o $@;
```



Taming Growing Complexity of Rules

```
ifdef ESMF_TAU
include /home/users/sameer/TAU/tau-2.13.6/ibm64/lib/Makefile.tau-
callpath-mpi-compensate-pdt
endif

...
.c.o:
ifdef PDTDIR
    -echo "Using TAU/PDT to instrument $<: Building .c.o"
    -$(PDTCPARSE) $< ${CFLAGS} ${CPPFLAGS} ${TAU_ESMC_INCLUDE}
${TAU_MPI_INCLUDE}
        -if [ -f $*.pdb ] ; then $(TAUINSTR) $*.pdb $< -o $*.inst.c -f
${TAU_SELECT_FILE} ; fi;
        -${CC} -c ${COPTFLAGS} ${CFLAGS} ${CCPPFLAGS} ${ESMC_INCLUDE}
${TAU_DEFS} ${TAU_INCLUDE} ${TAU_MPI_INCLUDE} $*.inst.c
        if [ ! -f $*.o ] ; then ${CC} -c ${COPTFLAGS} ${CFLAGS} ${CCPPFLAGS}
${ESMC_INCLUDE} $< ; fi ;
else
    ${CC} -c ${COPTFLAGS} ${CFLAGS} ${CCPPFLAGS} ${ESMC_INCLUDE} $<
endif
```



AutoInstrumentation using TAU_COMPILER

- \$(TAU_COMPILER) stub Makefile variable (v2.13.7+)
- Invokes PDT parser, TAU instrumentor, compiler through **tau_compiler.sh** shell script
- Requires minimal changes to application Makefile
 - Compilation rules are not changed
 - User adds \$(TAU_COMPILER) before compiler name
 - F90=mpxlf90
 - Changes to
 - F90= **\$(TAU_COMPILER)** mpxlf90
- Passes options from TAU stub Makefile to the four compilation stages
- Uses original compilation command if an error occurs



TAU_COMPILER Commandline Options

- See `<taudir>/<arch>/bin/tau_compiler.sh -help`
- Compilation:

```
% mpixlf90 -c foo.f90
```

Changes to

```
% f95parse foo.f90 $ (OPT1)
% tau_instrumentor foo.pdb foo.f90
              -o foo.inst.f90 $ (OPT2)
% mpixlf90 -c foo.f90 $ (OPT3)
```

- Linking:

```
% mpixlf90 foo.o bar.o -o app
```

Changes to

```
% mpixlf90 foo.o bar.o -o app $ (OPT4)
```

- Where options OPT[1-4] default values may be overridden by the user:

```
F90 = $(TAU_COMPILER) $(MYOPTIONS) mpixlf90
```



Using TAU_COMPILER

```
include /usr/common/acts/TAU/tau-2.13.7/rs6000/lib/  
Makefile.tau-mpi-pdt
```

```
F90 = $(TAU_COMPILER) mp xl f90  
OBJS = f1.o f2.o f3.o ...  
LIBS = -Lappdir -lapplib  
  
app: $(OBJS)  
    $(F90) $(OBJS) -o app $(LIBS)  
  
.f90.o:  
    $(F90) -c $<
```



Overriding Default Options: TAU_COMPILER

```
include /usr/common/acts/TAU/tau-2.13.7/rs6000/lib/  
        Makefile.tau-mpi-pdt-trace  
  
MYOPTIONS= -optVerbose -optKeepFiles  
  
F90 = $(TAU_COMPILER) $(MYOPTIONS) mpixlf90  
OBJS = f1.o f2.o f3.o ...  
LIBS = -Lappdir -lapplib1 -lapplib2 ...  
  
app: $(OBJS)  
      $(F90) $(OBJS) -o app $(LIBS)  
  
.f90.o:  
      $(F90) -c $<
```



Using PDT: *tau_instrumentor*

```
% tau_instrumentor
Usage : tau_instrumentor < pdbfile > < sourcefile > [ -o < outputfile > ] [ -noinline ]
[ -g groupname ] [ -i headerfile ] [ -c | -c++ | -fortran ] [ -f < instr_req_file > ]
For selective instrumentation, use -f option
% tau_instrumentor foo.pdb foo.cpp -o foo.inst.cpp -f selective.dat
% cat selective.dat
# Selective instrumentation: Specify an exclude/include list of routines/files.

BEGIN_EXCLUDE_LIST
void quicksort(int *, int, int)
void sort_5elements(int *)
void interchange(int *, int *)
END_EXCLUDE_LIST

BEGIN_FILE_INCLUDE_LIST
Main.cpp
Foo?.c
*.C
END_FILE_INCLUDE_LIST
# Instruments routines in Main.cpp, Foo?.c and *.C files only
# Use BEGIN_[FILE]_INCLUDE_LIST with END_[FILE]_INCLUDE_LIST
```



tau_reduce: Rule-Based Overhead Analysis

- Analyze the performance data to determine events with high (relative) overhead performance measurements
- Create a select list for excluding those events
- Rule grammar (used in *tau_reduce* tool)

[GroupName:] Field Operator Number

- *GroupName* indicates rule applies to events in group
- *Field* is a event metric attribute (from profile statistics)
 - numcalls, numsubs, percent, usec, cumusec, count [PAPI], totalcount, stdev, usecs/call, counts/call
- *Operator* is one of >, <, or =
- *Number* is any number
- Compound rules possible using & between simple rules



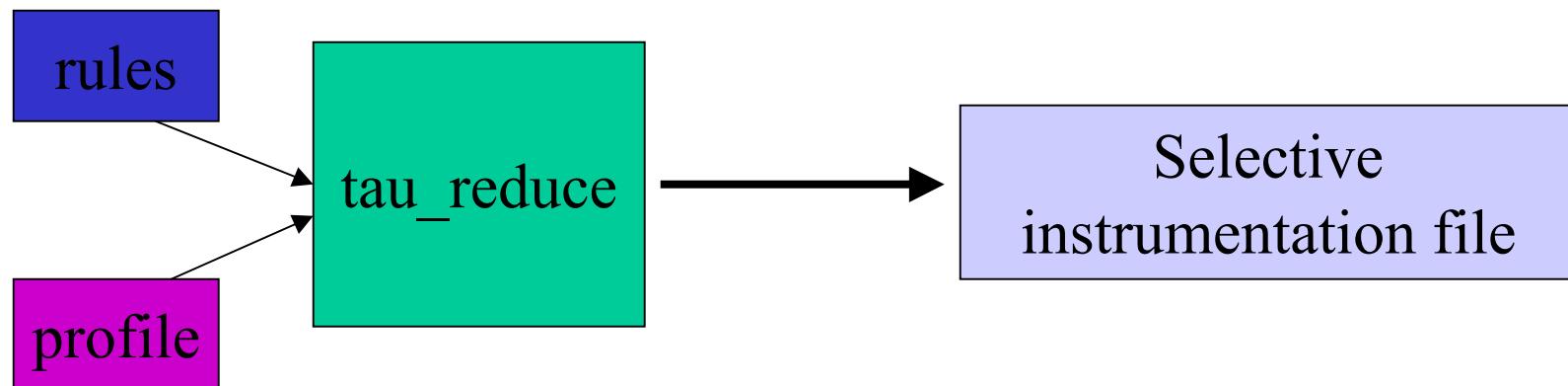
Example Rules

- #Exclude all events that are members of TAU_USER
#and use less than 1000 microseconds
TAU_USER:usec < 1000
- #Exclude all events that have less than 100
#microseconds and are called only once
usec < 1000 & numcalls = 1
- #Exclude all events that have less than 1000 usecs per
#call OR have a (total inclusive) percent less than 5
usecs/call < 1000
percent < 5
- Scientific notation can be used
 - **usec>1000 & numcalls>400000 & usecs/call<30 & percent>25**

TAU_REDUCE



- Reads profile files and rules
- Creates selective instrumentation file
 - Specifies which routines should be excluded from instrumentation





Using TAU – A tutorial

- Configuration
- Instrumentation
 - Manual
 - MPI – Wrapper interposition library
 - PDT- Source rewriting for C,C++, F77/90/95
 - ○ OpenMP – Directive rewriting
 - Component based instrumentation – Proxy components
 - Binary Instrumentation
 - DyninstAPI – Runtime Instrumentation/Rewriting binary
 - Java – Runtime instrumentation
 - Python – Runtime instrumentation
- Measurement
- Performance Analysis



Using Opari with TAU

Step I: Configure KOJAK/opari [Download from <http://www.fz-juelich.de/zam/kojak/>]

```
% cd kojak-1.0; cp mf/Makefile.defs.ibm Makefile.defs;  
edit Makefile  
% make
```

Builds opari

Step II: Configure TAU with Opari (used here with MPI and PDT)

```
% configure -opari=/usr/contrib/TAU/kojak-1.0/opari  
-mpiinc=/usr/lpp/ppe.poe/include  
-mpilib=/usr/lpp/ppe.poe/lib  
-pdt=/usr/contrib/TAU/pdtoolkit-3.2.1  
% make clean; make install
```

Instrumentation of OpenMP Constructs



- OpenMP Pragma And Region Instrumentor
- Source-to-Source translator to insert POMP calls around OpenMP constructs and API functions
- Done: Supports
 - Fortran77 and Fortran90, OpenMP 2.0
 - C and C++, OpenMP 1.0
 - POMP Extensions
 - EPILOG and TAU POMP implementations
 - Preserves source code information (`#line line file`)
- Work in Progress:
 - Investigating standardization through OpenMP Forum



OpenMP API Instrumentation

□ Transform

- `omp_##_lock()` → `pomp_##_lock()`
- `omp_##_nest_lock()` → `pomp_##_nest_lock()`

[# = `init` | `destroy` | `set` | `unset` | `test`]

□ POMP version

- Calls omp version internally
- Can do extra stuff before and after call



Example: !\$OMP PARALLEL DO Instrumentation

```
call pomp_parallel_fork(d)
!$OMP PARALLEL other-clauses...
    call pomp_parallel_begin(d)
    call pomp_do_enter(d)
        !$OMP DO schedule-clauses, ordered-clauses,
                  lastprivate-clauses
            do loop
        !$OMP END DO NOWAIT
        call pomp_barrier_enter(d)
        !$OMP BARRIER
        call pomp_barrier_exit(d)
        call pomp_do_exit(d)
        call pomp_parallel_end(d)
    !$OMP END PARALLEL DO
call pomp_parallel_join(d)
```



Opari Instrumentation: Example

□ OpenMP directive instrumentation

```
pomp_for_enter(&omp_rd_2);
#line 252 "stommel.c"
#pragma omp for schedule(static) reduction(+: diff) private(j)
    firstprivate (a1,a2,a3,a4,a5) nowait
for( i=i1;i<=i2;i++ ) {
    for(j=j1;j<=j2;j++) {
        new_psi[i][j]=a1*psi[i+1][j] + a2*psi[i-1][j] + a3*psi[i][j+1]
            + a4*psi[i][j-1] - a5*the_for[i][j];
        diff=diff+fabs(new_psi[i][j]-psi[i][j]);
    }
}
pomp_barrier_enter(&omp_rd_2);
#pragma omp barrier
pomp_barrier_exit(&omp_rd_2);
pomp_for_exit(&omp_rd_2);
#line 261 "stommel.c"
```



OPARI: Makefile Template (Fortran)

```
OMP77 = ...          # insert f77 OpenMP compiler here
OMP90 = ...          # insert f90 OpenMP compiler here

.f.o:
    opari $<
    $(OMP77) $(CFLAGS) -c $*.mod.F

.f90.o:
    opari $<
    $(OMP90) $(CXXFLAGS) -c $*.mod.F90

opari.init:
    rm -rf opari.rc

opari.tab.o:
    opari -table opari.tab.c
    $(CC) -c opari.tab.c

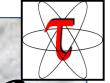
myprog: opari.init myfile*.o ... opari.tab.o
        $(OMP90) -o myprog myfile*.o opari.tab.o $(TAU_LIBS)

myfile1.o: myfile1.f90
myfile2.o: ...
```



CCA Performance Observation Component

- Common Component Architecture for Scientific Components [www.cca-forum.org]
- Design measurement port and measurement interfaces
 - Timer
 - start/stop
 - set name/type/group
 - Control
 - enable/disable groups
 - Query
 - get timer names
 - metrics, counters, dump to disk
 - Event
 - user-defined events



CCA C++ (CCAFFEINE) Performance Interface

```
namespace performance {
namespace ccaports {
    class Measurement: public virtual classic::gov::cca::Port {
        public:
            virtual ~Measurement () {}

            /* Create a Timer interface */
            virtual performance::Timer* createTimer(void) = 0;
            virtual performance::Timer* createTimer(string name) = 0;
            virtual performance::Timer* createTimer(string name, string type) = 0;
            virtual performance::Timer* createTimer(string name, string type,
                string group) = 0;

            /* Create a Query interface */
            virtual performance::Query* createQuery(void) = 0;

            /* Create a user-defined Event interface */
            virtual performance::Event* createEvent(void) = 0;
            virtual performance::Event* createEvent(string name) = 0;

            /* Create a Control interface for selectively enabling and disabling
             * the instrumentation based on groups */
            virtual performance::Control* createControl(void) = 0;
    };
}
}
```

Measurement port

Measurement interfaces



CCA Timer Interface Declaration

```
namespace performance {
    class Timer {
        public:
            virtual ~Timer() {}

            /* Implement methods in a derived class to provide functionality */

            /* Start and stop the Timer */
            virtual void start(void) = 0;
            virtual void stop(void) = 0;

            /* Set name and type for Timer */
            virtual void setName(string name) = 0;
            virtual string getName(void) = 0;
            virtual void setType(string name) = 0;
            virtual string getType(void) = 0;

            /* Set the group name and group type associated with the Timer */
            virtual void setGroupName(string name) = 0;
            virtual string getGroupName(void) = 0;
            virtual void setGroupId(unsigned long group ) = 0;
            virtual unsigned long getGroupId(void) = 0;
    };
}
```

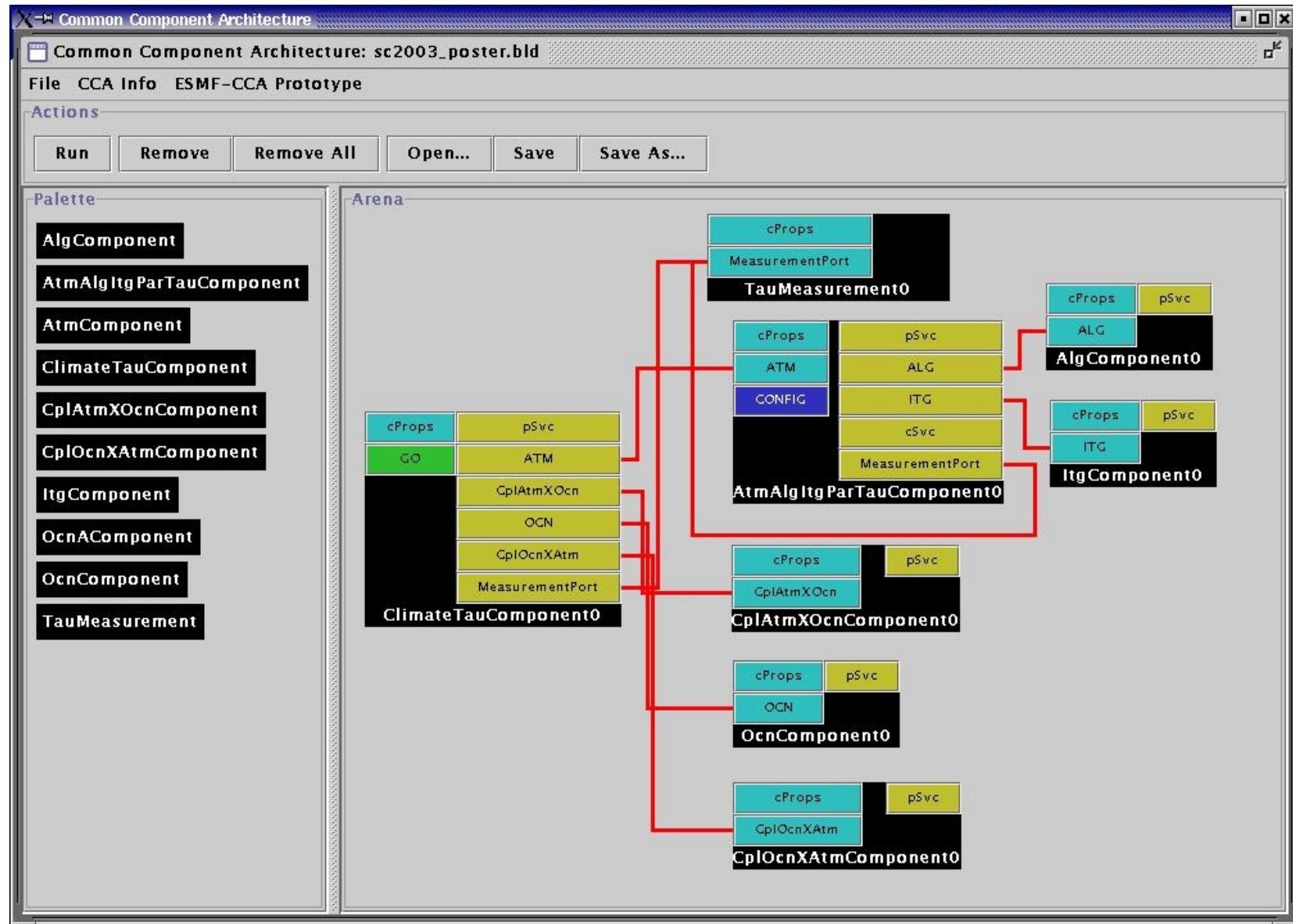
Timer interface methods



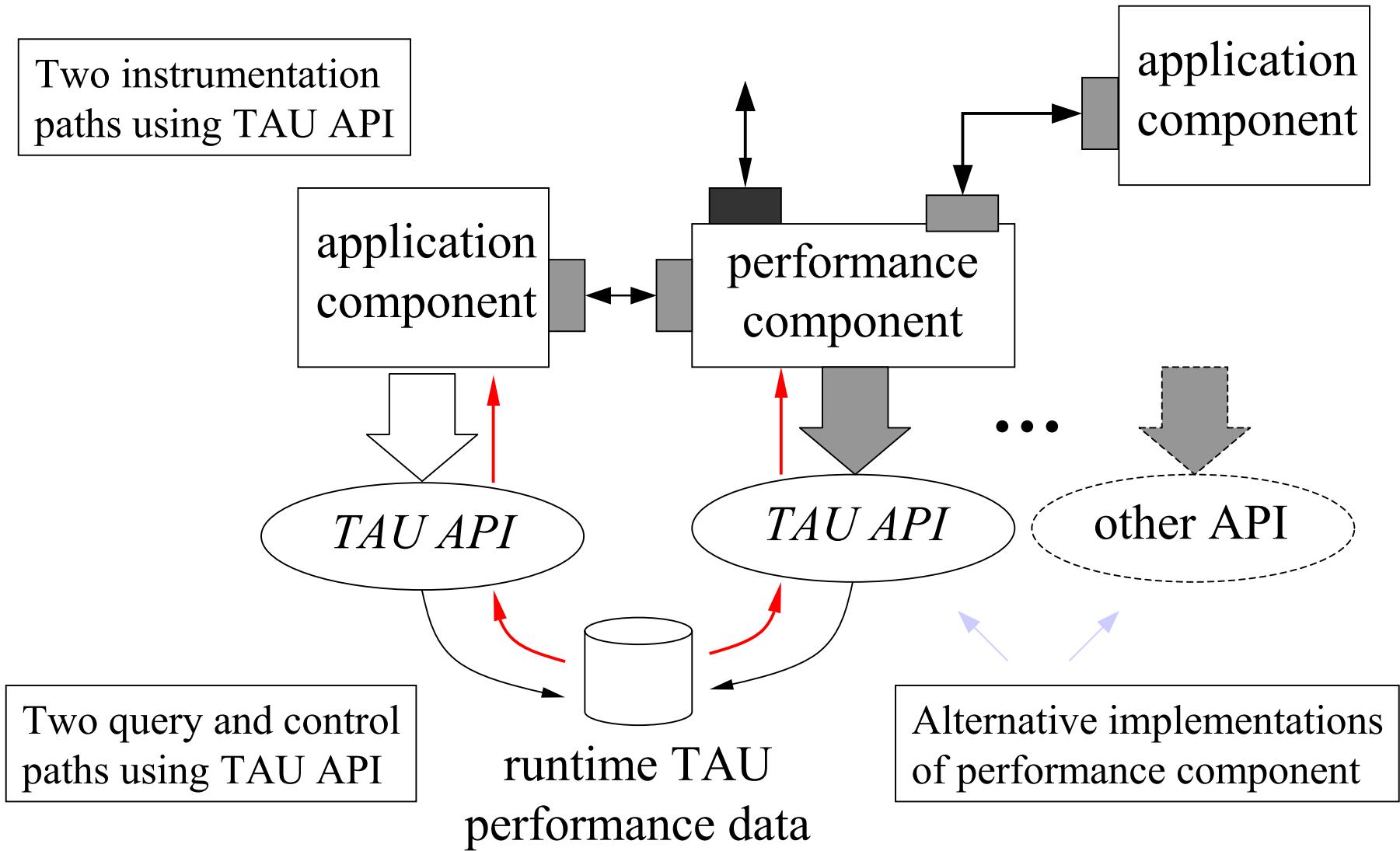
Use of Observation Component in CCA Example

```
#include "ports/Measurement_CCA.h"
...
double MonteCarloIntegrator::integrate(double lowBound, double upBound,
                                         int count) {
    classic::gov::cca::Port * port;
    double sum = 0.0;
    // Get Measurement port
    port = frameworkServices->getPort ("MeasurementPort");
    if (port)
        measurement_m = dynamic_cast < performance::ccaports::Measurement * >(port);
    if (measurement_m == 0){
        cerr << "Connected to something other than a Measurement port";
        return -1;
    }
    static performance::Timer* t = measurement_m->createTimer(
        string("IntegrateTimer"));
    t->start();
    for (int i = 0; i < count; i++) {
        double x = random_m->getRandomNumber ();
        sum = sum + function_m->evaluate (x);
    }
    t->stop();
}
```

Using TAU Component in ESMF/CCA [S. Zhou]



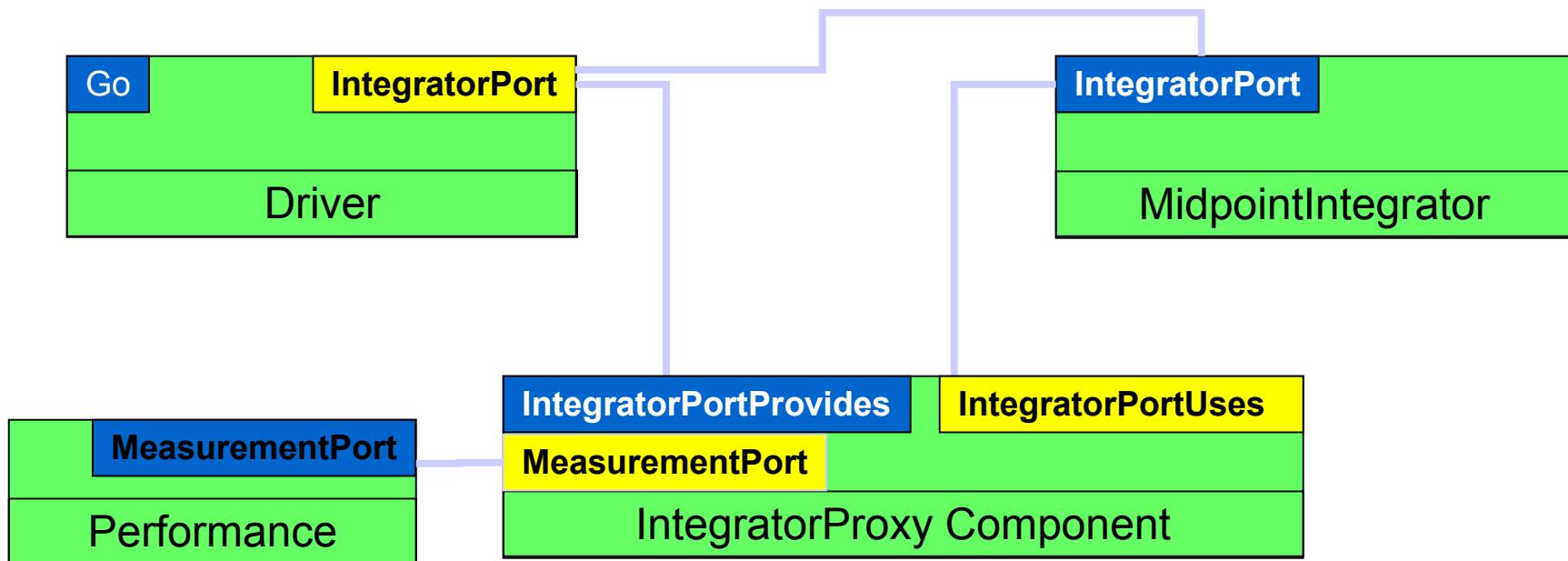
What's Going On Here?





Proxy Component

- Interpose a proxy component for each port
- Inside the proxy, track caller/callee invocations, timings
- Automate the process of proxy component creation
 - Using PDT for static analysis of components



Dynamic Instrumentation



- TAU uses DyninstAPI for runtime code patching
- *tau_run* (mutator) loads measurement library
- Instruments mutatee
- MPI issues:
 - one mutator per executable image [TAU, DynaProf]
 - one mutator for several executables [Paradyn, DPCL]



Using DyninstAPI with TAU

Step I: Install DyninstAPI[Download from <http://www.dyninst.org>]

```
% cd dyninstAPI-4.0.2/core; make
```

Set DyninstAPI environment variables (including LD_LIBRARY_PATH)

Step II: Configure TAU with Dyninst

```
% configure -dyninst=/usr/local/dyninstAPI-4.0.2  
% make clean; make install
```

Builds <taudir>/<arch>/bin/tau_run

```
% tau_run [<-o outfile>] [-Xrun<libname>]  
[-f <select_inst_file>] [-v] <infile>  
% tau_run -o a.inst.out a.out
```

Rewrites a.out

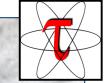
```
% tau_run klargest
```

Instruments klargest with TAU calls and executes it

```
% tau_run -XrunTAUsh-papi a.out
```

Loads libTAUsh-papi.so instead of libTAU.so for measurements

NOTE: All compilers and platforms are not yet supported (work in progress)



Virtual Machine Performance Instrumentation

- Integrate performance system with VM
 - Captures robust performance data (e.g., thread events)
 - Maintain features of environment
 - portability, concurrency, extensibility, interoperation
 - Allow use in optimization methods
- JVM Profiling Interface (JVMPPI)
 - Generation of JVM events and hooks into JVM
 - Profiler agent (TAU) loaded as shared object
 - registers events of interest and address of callback routine
 - Access to information on dynamically loaded classes
 - No need to modify Java source, bytecode, or JVM



Using TAU with Java Applications

Step I: Sun JDK 1.2+ [download from www.javasoft.com]

Step II: Configure TAU with JDK (v 1.2 or better)

```
% configure -jdk=/usr/java2 -TRACE -PROFILE  
% make clean; make install
```

Builds <taudir>/<arch>/lib/libTAU.so

For Java (without instrumentation):

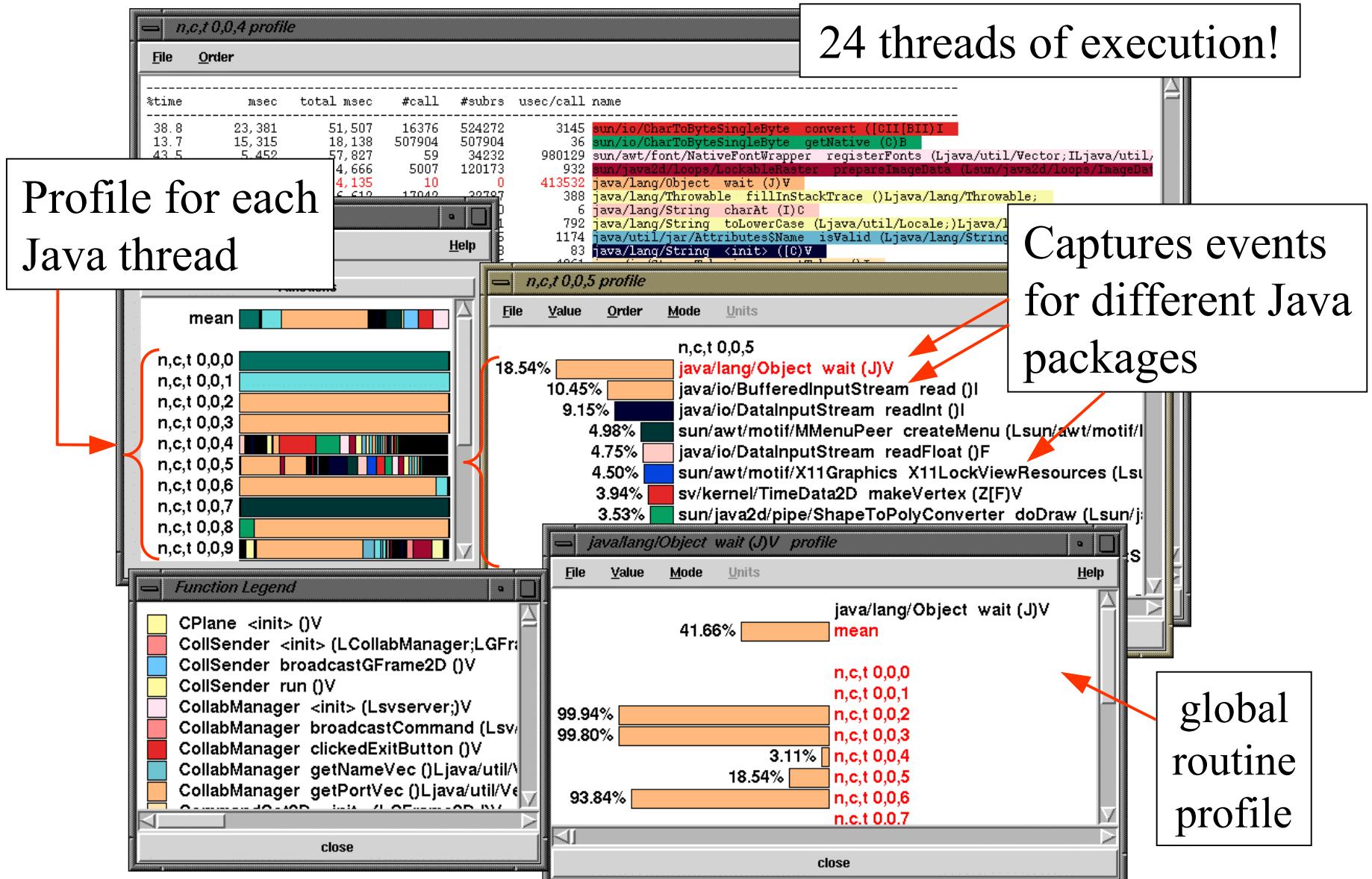
```
% java application
```

With instrumentation:

```
% java -XrunTAU application  
% java -XrunTAU:exclude=sun/io,java application
```

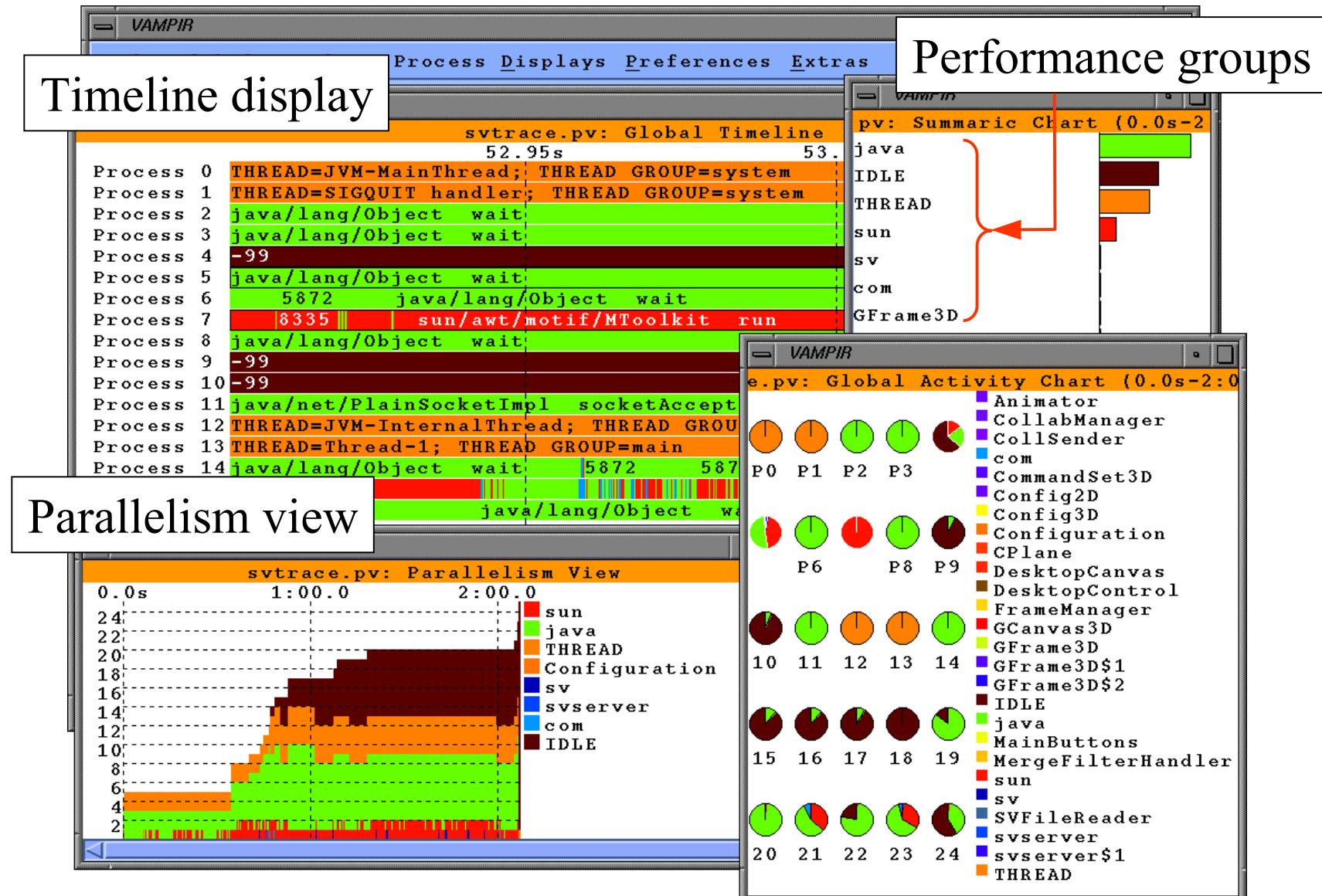
Excludes sun/io/* and java/* classes

TAU Profiling of Java Application (SciVis)



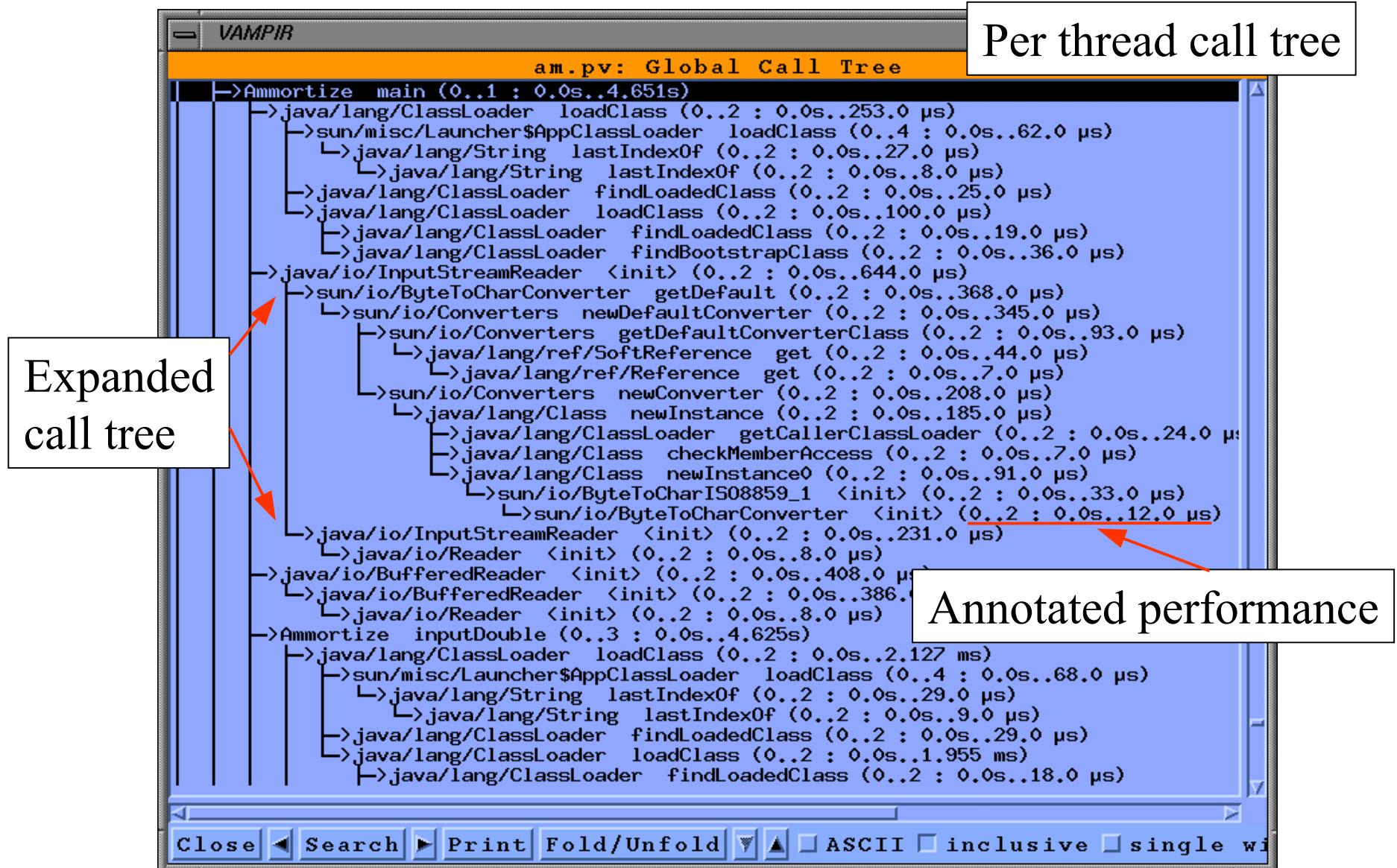


TAU Tracing of Java Application (SciVis)





Vampir Dynamic Call Tree View (SciVis)





Using TAU with Python Applications

Step I: Configure TAU with Python

```
% configure --pythoninc=/usr/include/python2.2/include  
% make clean; make install
```

Builds <taudir>/<arch>/lib/<bindings>/pytau.py and tau.py packages
for manual and automatic instrumentation respectively

```
% setenv PYTHONPATH $PYTHONPATH\:<taudir>/<arch>/lib/ [<dir>]
```



Python Automatic Instrumentation Example

```
#!/usr/bin/env/python

import tau
from time import sleep

def f2():
    print " In f2: Sleeping for 2 seconds "
    sleep(2)
def f1():
    print " In f1: Sleeping for 3 seconds "
    sleep(3)

def OurMain():
    f1()
tau.run('OurMain()')
```

Running:

```
% setenv PYTHONPATH <tau>/<arch>/lib
% ./auto.py
Instruments OurMain, f1, f2, print...
```



TAU Performance Measurement

- TAU supports profiling and tracing measurement
- TAU supports tracking application memory utilization
- Robust timing and hardware performance support using PAPI
- Support for online performance monitoring
 - Profile and trace performance data export to file system
 - Selective exporting
- Extension of TAU measurement for multiple counters
 - Creation of user-defined TAU counters
 - Access to system-level metrics
- Support for callpath measurement
- Integration with system-level performance data



Memory Profiling in TAU

Configuration option –**PROFILEMEMORY**

- Records global heap memory utilization for each function
- Takes one sample at beginning of each function and associates the sample with function name
- Independent of instrumentation/measurement options selected
- No need to insert macros/calls in the source code
- User defined atomic events appear in profiles/traces
- For Traces, see Vampir's Global Displays->CounterTimeline to view memory samples



Memory Profiling in TAU

- Instrumentation based observation of global heap memory (not per function)
 - call TAU_TRACK_MEMORY()
 - Triggers one sample every 10 secs
 - call TAU_TRACK_MEMORY_HERE()
 - Triggers sample at a specific location in source code
 - call TAU_SET_INTERRUPT_INTERVAL(seconds)
 - To set inter-interrupt interval for sampling
 - call TAU_DISABLE_TRACKING_MEMORY()
 - To turn off recording memory utilization
 - call TAU_ENABLE_TRACKING_MEMORY()
 - To re-enable tracking memory utilization



Using TAU's Malloc Wrapper Library for C/C++

```
include /usr/common/acts/TAU/tau-2.13.7/rs6000/lib/Makefile.tau-pdt
CC=$ (TAU_CC)
CFLAGS=$ (TAU_DEFS) $ (TAU_INCLUDE) $ (TAU_MEMORY_INCLUDE)
LIBS = $ (TAU_LIBS)
OBJS = f1.o f2.o ...
TARGET= a.out
TARGET: $ (OBJS)
        $ (F90) $ (LDFLAGS) $ (OBJS) -o $@ $ (LIBS)
.c.o:
        $ (CC) $ (CFLAGS) -c $< -o $@
```



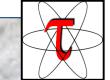
TAU's malloc/free wrapper for C/C++

```
#include <TAU.h>
#include <malloc.h>

int main(int argc, char **argv)
{
    TAU_PROFILE("int main(int, char **)", " ", TAU_DEFAULT);

    int *ary = (int *) malloc(sizeof(int) * 4096);

    // TAU's malloc wrapper library replaces this call automatically
    // when $(TAU_MEMORY_INCLUDE) is used in the Makefile.
    ...
    free(ary);
    // other statements in foo ...
}
```



Using TAU's Malloc Wrapper Library for C/C++

NumSamples	MaxValue	MinValue	MeanValue	name
1	40016.0	40016.0	40016.0	malloc size <file=main.cpp, line=252>
1	40016.0	40016.0	40016.0	free size <file=main.cpp, line=298>
12	30000.0	240.0	5590.0	malloc size <file=select.cpp, line=80>
12	30000.0	240.0	5590.0	malloc size <file=select.cpp, line=81>
3	30000.0	6000.0	17000.0	free size <file=select.cpp, line=107>
3	30000.0	6000.0	17000.0	free size <file=select.cpp, line=109>
1	8000.0	8000.0	8000.0	malloc size <file=main.cpp, line=258>
1	8000.0	8000.0	8000.0	free size <file=main.cpp, line=299>
7	6000.0	600.0	2228.5714	free size <file=select.cpp, line=118>
7	6000.0	600.0	2228.5714	free size <file=select.cpp, line=119>
2	240.0	240.0	240.0	free size <file=select.cpp, line=126>
2	240.0	240.0	240.0	free size <file=select.cpp, line=128>



Performance Mapping

- Associate performance with “significant” entities (events)
- Source code points are important
 - Functions, regions, control flow events, user events
- Execution process and thread entities are important
- Some entities are more abstract, harder to measure



Performance Mapping in Callpath Profiling

- Consider callgraph (callpath) profiling
 - Measure time (metric) along an edge (path) of callgraph
 - Incident edge gives parent / child view
 - Edge sequence (path) gives parent / descendant view
- Callpath profiling when callgraph is unknown
 - Must determine callgraph dynamically at runtime
 - Map performance measurement to dynamic call path state
- Callpath levels
 - 1-level: current callgraph node/flat profile
 - 2-level: immediate parent (descendant)
 - k -level: k th nodes in the calling path



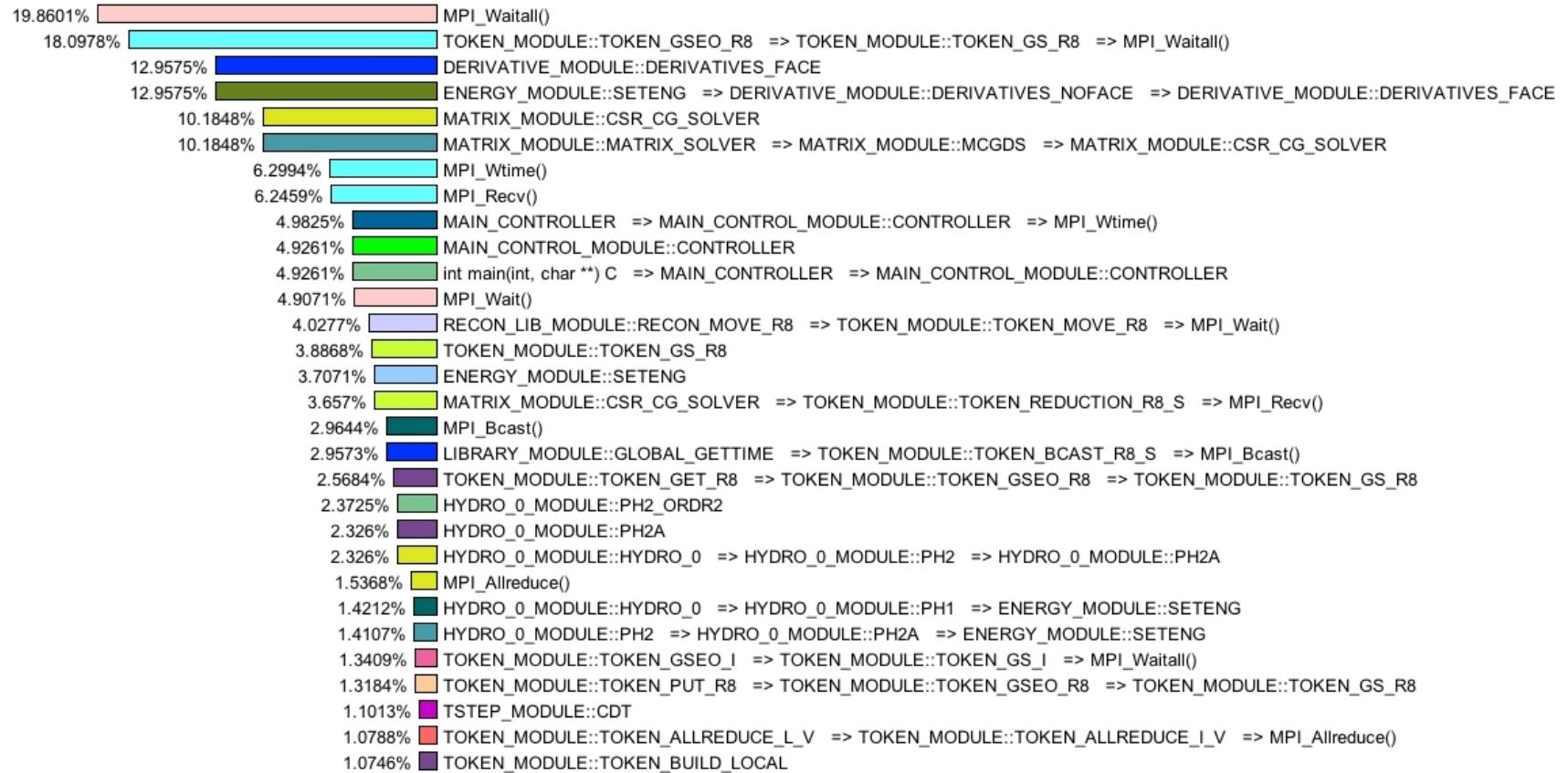
k-Level Callpath Implementation in TAU

- TAU maintains a performance event (routine) callstack
- Profiled routine (child) looks in callstack for parent
 - Previous profiled performance event is the parent
 - A *callpath profile structure* created first time parent calls
 - TAU records parent in a *callgraph map* for child
 - String representing k-level callpath used as its key
 - “**a()**=>**b()**=>**c()**” : name for time spent in “c” when called by “b” when “b” is called by “a”
- Map returns pointer to callpath profile structure
 - k-level callpath is profiled using this profiling data
 - Set environment variable **TAU_CALLPATH_DEPTH** to depth
- Build upon TAU’s performance mapping technology
- Measurement is independent of instrumentation
- Use **-PROFILECALLPATH** to configure TAU



k-Level Callpath Implementation in TAU

Metric Name: Time
Value Type: exclusive





Gprof Style Callpath View in Paraprof

Metric Name: Time
Sorted By: exclusive
Units: seconds

Exclusive	Inclusive	Calls/Tot.Calls	Name[id]
<hr/>			
1.8584	1.8584	1196/13188	TOKEN_MODULE::TOKEN_GS_I [521]
0.584	0.584	234/13188	TOKEN_MODULE::TOKEN_GS_L [544]
25.0819	25.0819	11758/13188	TOKEN_MODULE::TOKEN_GS_R8 [734]
--> 27.5242	27.5242	13188	MPI_Waitall() [525]
<hr/>			
17.9579	39.1657	156/156	DERIVATIVE_MODULE::DERIVATIVES_NOFACE [841]
--> 17.9579	39.1657	156	DERIVATIVE_MODULE::DERIVATIVES_FACE [843]
0.0156	0.0195	312/312	TIMER_MODULE::TIMERSET [77]
0.1133	9.1269	2340/2340	MESSAGE_MODULE::CLONE_GET_R8 [808]
0.1602	11.4608	4056/4056	MESSAGE_MODULE::CLONE_PUT_R8 [850]
0.0059	0.6006	117/117	MESSAGE_MODULE::CLONE_PUT_I [856]
<hr/>			
14.1151	21.6209	5/5	MATRIX_MODULE::MCGDS [1443]
--> 14.1151	21.6209	5	MATRIX_MODULE::CSR_CG_SOLVER [1470]
0.0654	1.2617	1005/1005	TOKEN_MODULE::TOKEN_GET_R8 [769]
0.0557	5.2714	1005/1005	TOKEN_MODULE::TOKEN_REDUCTION_R8_S [1475]
0.0703	0.9726	1000/1000	TOKEN_MODULE::TOKEN_REDUCTION_R8_V [208]



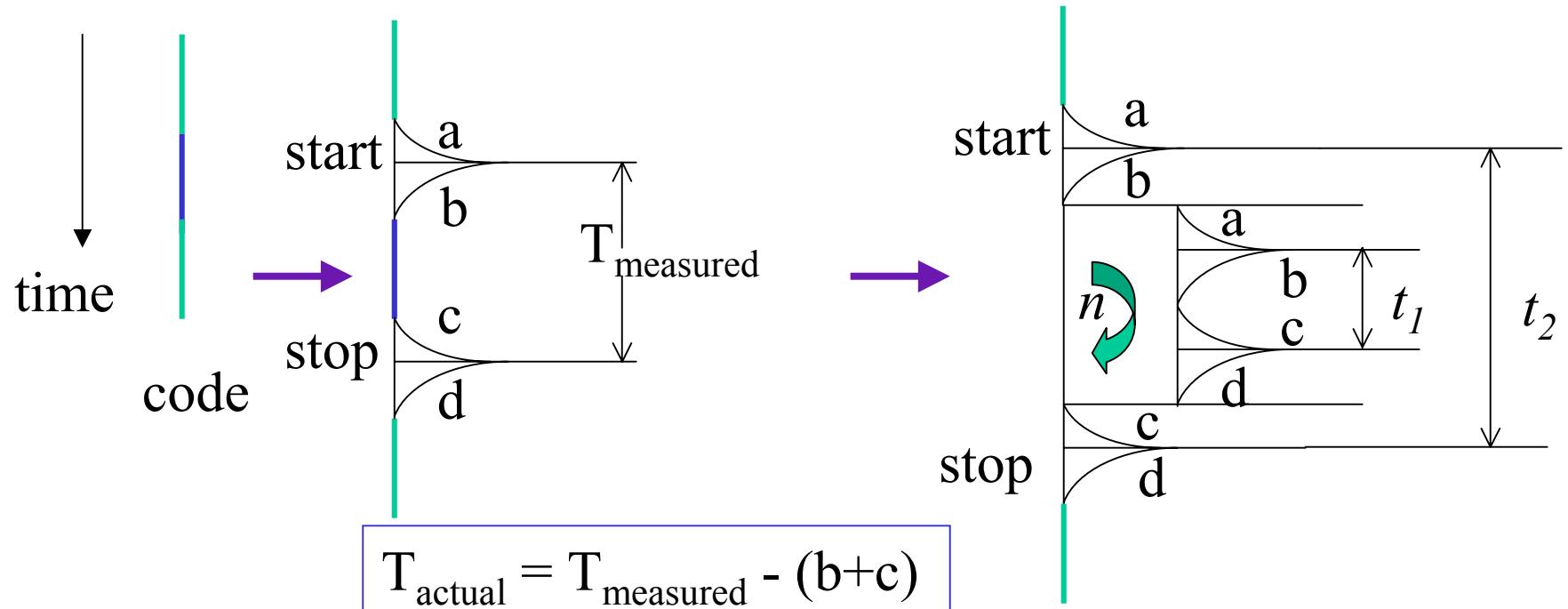
Compensation of Instrumentation Overhead

- Runtime estimation of a single timer overhead
- Evaluation of number of timer calls along a calling path
- Compensation by subtracting timer overhead
- Recalculation of performance metrics to improve the accuracy of measurements
- Configure TAU with **-COMPENSATE** configuration option



Estimating Timer Overheads

- Introduce a pair of timer calls (start/stop)



$$t_1 = n * (b+c)$$

$$t_2 = b + n * (a+b+c+d) + c$$

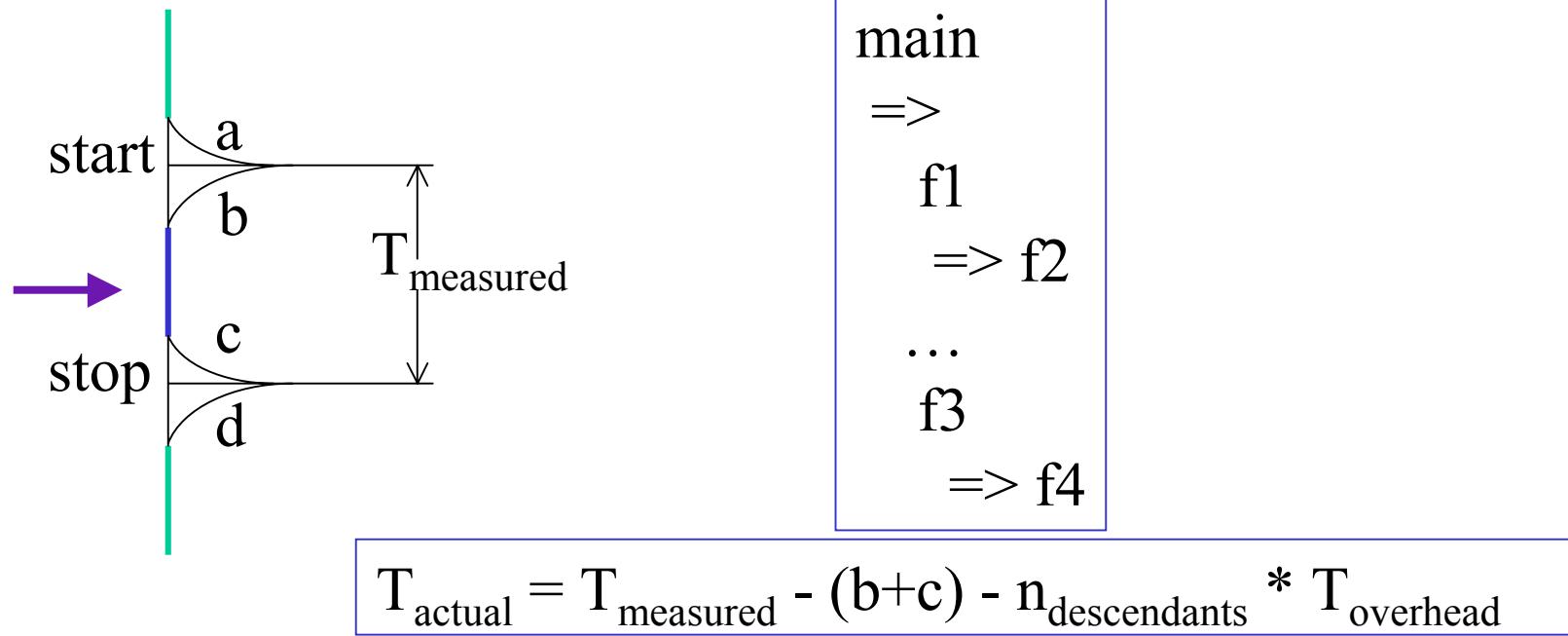
$$T_{overhead} = a+b+c+d = (t_2 - (t_1/n))/n$$

$$T_{null} = b+c = t_1/n$$



Recalculating Inclusive Time

- Number of children/grandchildren... nodes
- Traverse callstack





Grouping Performance Data in TAU

□ Profile Groups

- A group of related routines forms a profile group
- Statically defined
 - TAU_DEFAULT, TAU_USER[1-5], TAU_MESSAGE, TAU_IO, ...
- Dynamically defined
 - group name based on string, such as “**adlib**” or “**particles**”
 - runtime lookup in a map to get unique group identifier
 - uses *tau_instrumentor* to instrument
- Ability to change group names at runtime
- Group-based instrumentation and measurement control



- Parallel profile analysis
 - *Pprof*
 - parallel profiler with text-based display
 - *ParaProf*
 - Graphical, scalable, parallel profile analysis and display
- Trace analysis and visualization
 - Trace merging and clock adjustment (if necessary)
 - Trace format conversion (ALOG, SDDF, VTF, Paraver)
 - Trace visualization using *Vampir* (Pallas/Intel)



Pprof Output (NAS Parallel Benchmark – LU)

- Intel Quad PIII Xeon
- F90 + MPICH
- Profile
 - Node
 - Context
 - Thread
- Events
 - code
 - MPI

emacs@neutron.cs.uoregon.edu

Buffers Files Tools Edit Search Mule Help

Reading Profile files in profile.*

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	1	3:11.293	1	15	191293269	applu
99.6	3,667	3:10.463	3	37517	63487925	bcast_inputs
67.1	491	2:08.326	37200	37200	3450	exchange_1
44.5	6,461	1:25.159	9300	18600	9157	buts
41.0	1:18.436	1:18.436	18600	0	4217	MPI_Recv()
29.5	6,778	56,407	9300	18600	6065	blts
26.2	50,142	50,142	19204	0	2611	MPI_Send()
16.2	24,451	31,031	301	602	103096	rhs
3.9	7,501	7,501	9300	0	807	jacld
3.4	838	6,594	604	1812	10918	exchange_3
3.4	6,590	6,590	9300	0	709	jacu
2.6	4,989	4,989	608	0	8206	MPI_Wait()
0.2	0.44	400	1	4	400081	init_comm
0.2	398	399	1	39	399634	MPI_Init()
0.1	140	247	1	47616	247086	setiv
0.1	131	131	57252	0	2	exact
0.1	89	103	1	2	103168	erhs
0.1	0.966	96	1	2	96458	read_input
0.0	95	95	9	0	10603	MPI_Bcast()
0.0	26	44	1	7937	44878	error
0.0	24	24	608	0	40	MPI_Irecv()
0.0	15	15	1	5	15630	MPI_Finalize()
0.0	4	12	1	1700	12335	setbv
0.0	7	8	3	3	2893	l2norm
0.0	3	3	8	0	491	MPI_Allreduce()
0.0	1	3	1	6	3874	pintgr
0.0	1	1	1	0	1007	MPI_Barrier()
0.0	0.116	0.837	1	4	837	exchange_4
0.0	0.512	0.512	1	0	512	MPI_Keyval_create()
0.0	0.121	0.353	1	2	353	exchange_5
0.0	0.024	0.191	1	2	191	exchange_6
0.0	0.103	0.103	6	0	17	MPI_Type_contiguous()

--:-- NPB_LU.out (Fundamental)--L8--Top----



Terminology – Example

- For routine “int main()”:
- Exclusive time
 - $100 - 20 - 50 - 20 = 10$ secs
- Inclusive time
 - 100 secs
- Calls
 - 1 call
- Subrs (no. of child routines called)
 - 3
- Inclusive time/call
 - 100secs

```
int main( )
{ /* takes 100 secs */

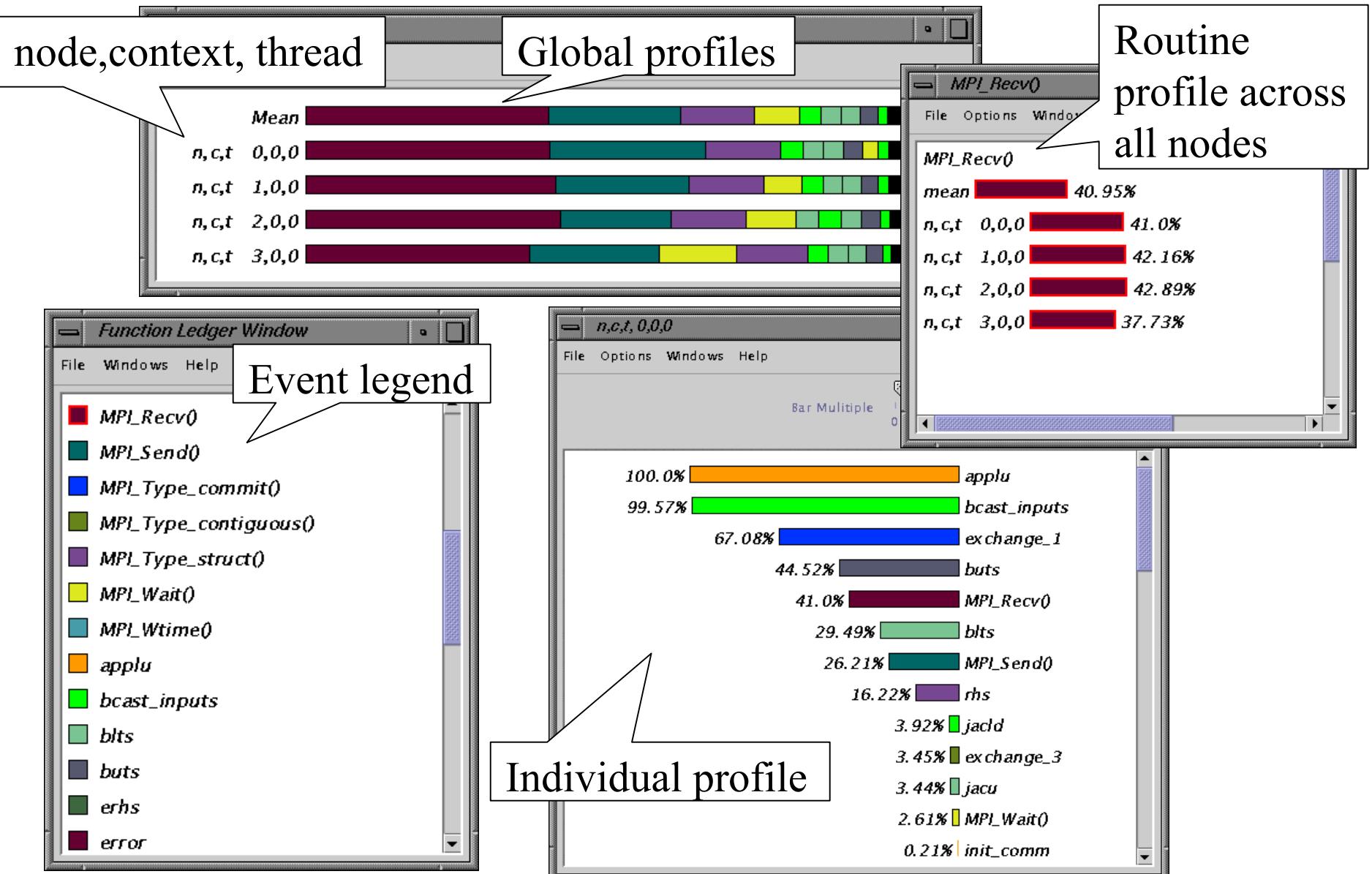
    f1(); /* takes 20 secs */
    f2(); /* takes 50 secs */
    f1(); /* takes 20 secs */

    /* other work */
}

/*
Time can be replaced by counts
from PAPI e.g., PAPI_FP_INS. */
```



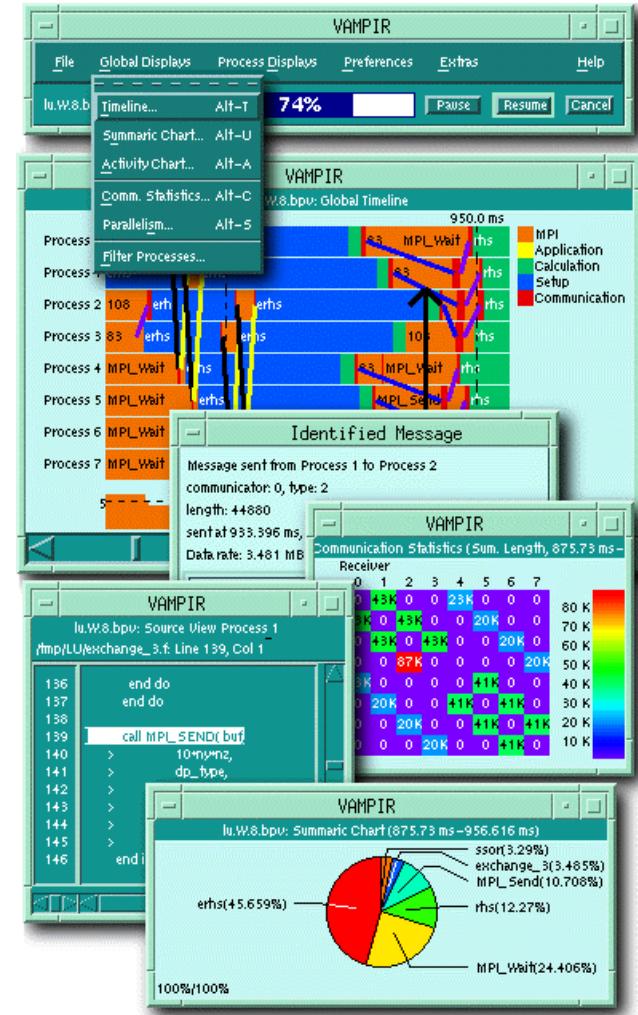
ParaProf (NAS Parallel Benchmark – LU)



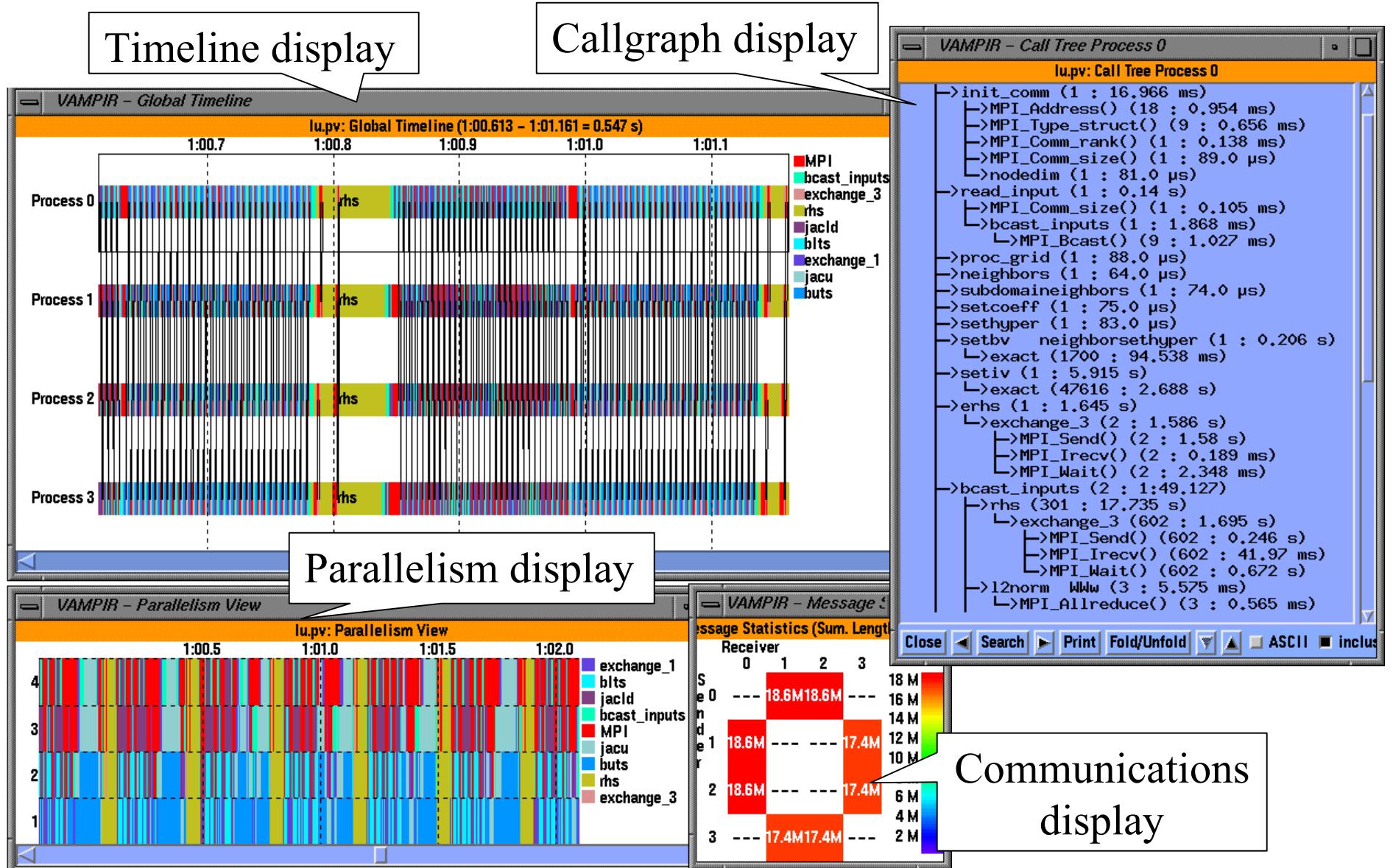
Intel Trace Analyzer/Vampir Trace Visualizer



- Visualization and Analysis of MPI Programs
- Originally developed by Forschungszentrum Jülich
- Current development by Technical University Dresden, Germany
- Distributed by Intel
- <http://www.pallas.de/pages/vampir.htm>

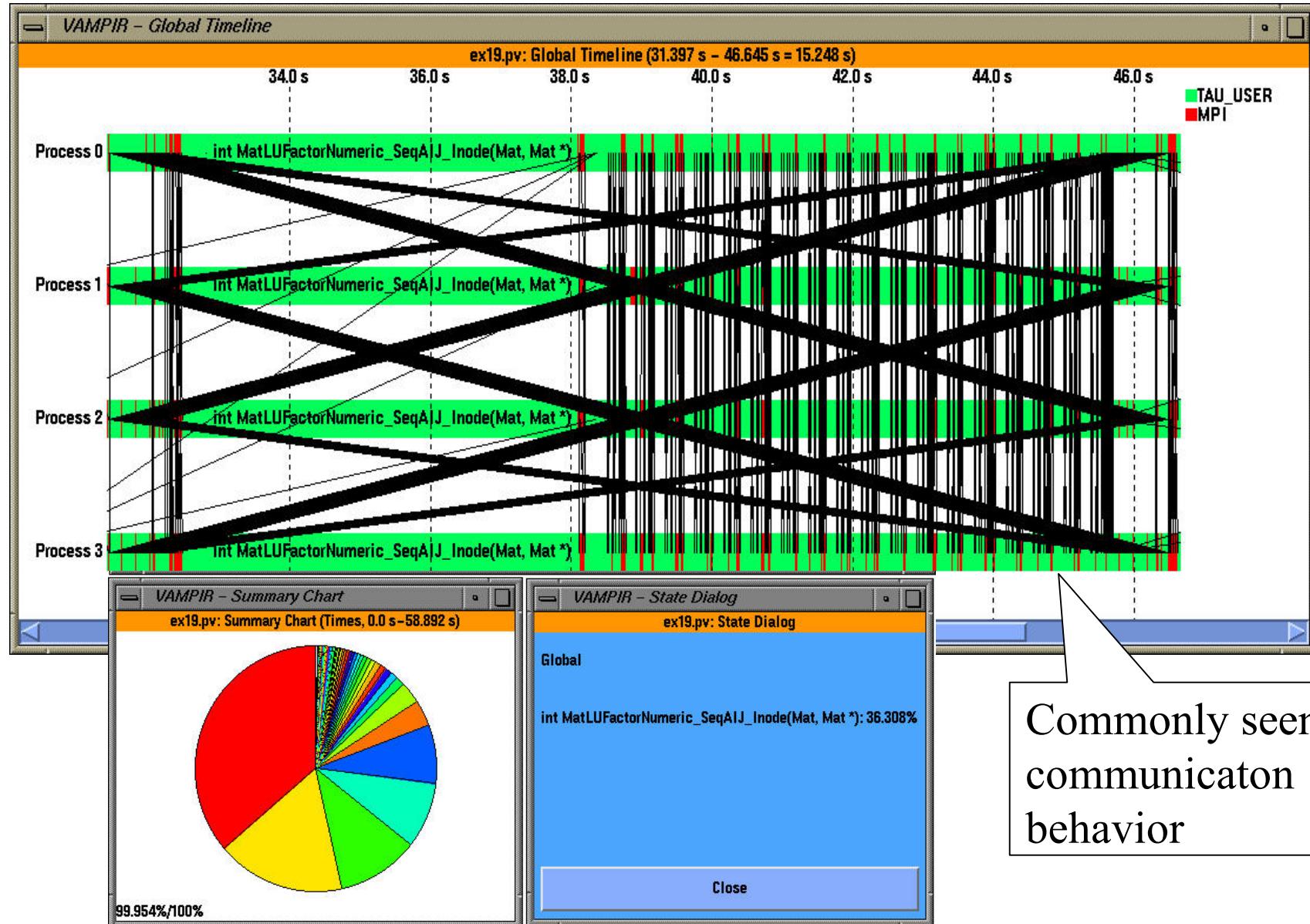


TAU + Vampir (NAS Parallel Benchmark – LU)



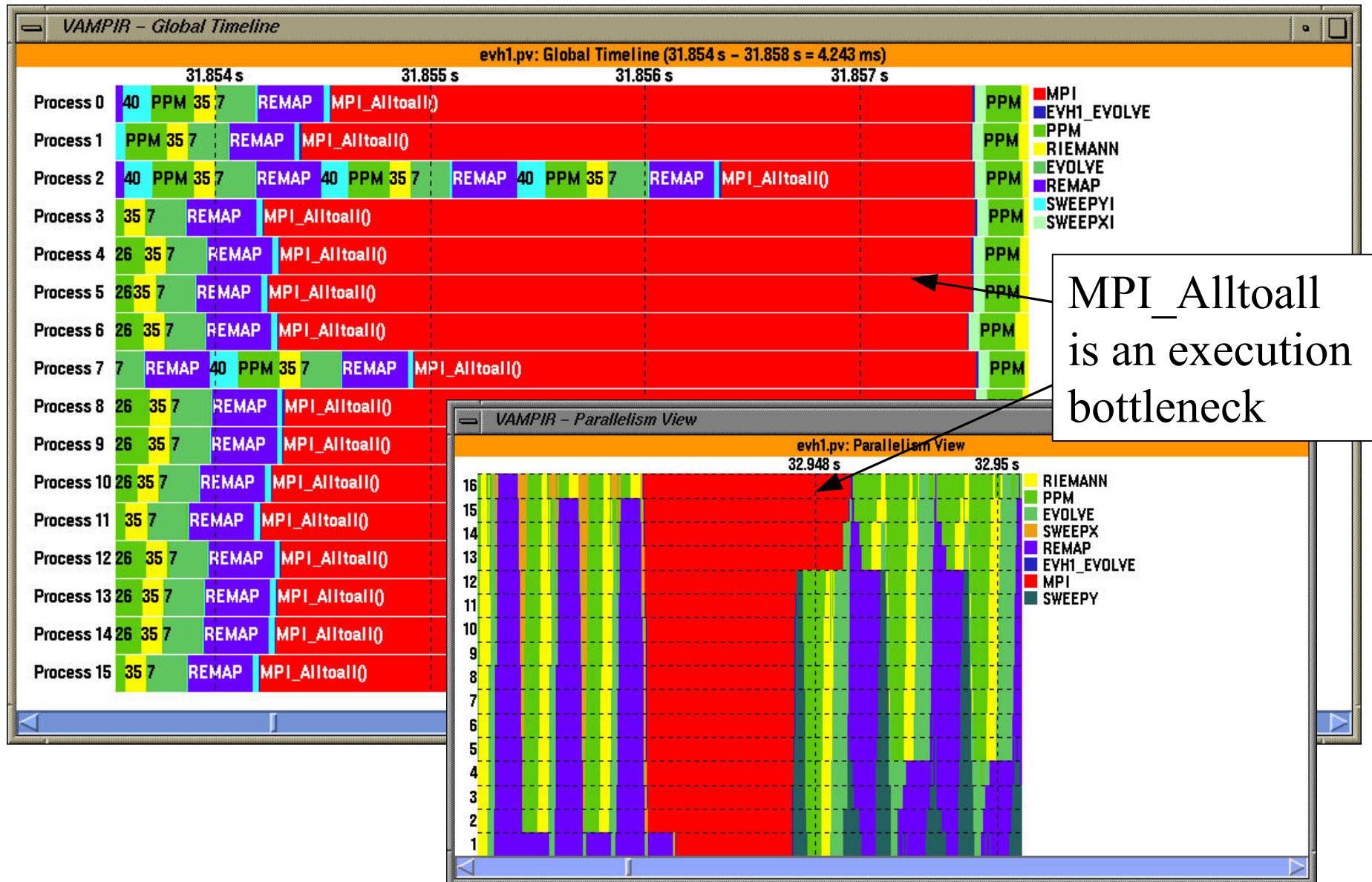


PETSc ex19 (Tracing)





TAU's EVH1 Execution Trace in Vampir





Using TAU with Vampir

- Configure TAU with -TRACE -vtf=dir option

```
% configure -TRACE -vtf=<dir>  
-MULTIPLECOUNTERS -papi=<dir> -mpi  
-pdt=dir ...
```

- Set environment variables

```
% setenv TAU_TRACEFILE foo.vpt.gz  
% setenv COUNTER1 GET_TIME_OF_DAY (reqd)  
% setenv COUNTER2 PAPI_FP_INS...
```

- Execute application (automatic merge/convert)

```
% poe a.out -procs 4  
% vampir foo.vpt.gz
```



Using TAU with Vampir

```
include /usr/common/acts/TAU/tau-
2.13.7/rs6000/lib/Makefile.tau-mpi-pdt-trace
F90 = $(TAU_F90)
LIBS = $(TAU_MPI_LIBS) $(TAU_LIBS) $(TAU_CXXLIBS)
OBJS = ...
TARGET= a.out
TARGET: $(OBJS)
        $(CXX) $(LDFLAGS) $(OBJS) -o $@ $(LIBS)
.f.o:
        $(F90) $(FFLAGS) -c $< -o $@
```



Using TAU with Vampir

```
% llsubmit job.sh  
% ls *.trc *.edf
```

Merging Trace Files

```
% tau_merge tau*.trc app.trc
```

Converting TAU Trace Files to Vampir and Paraver Trace formats

```
% tau_convert -pv app.trc tau.edf app.pv  
(use -vampir if application is multi-threaded)
```

```
% vampir app.pv
```

```
% tau_convert -paraver app.trc tau.edf app.par  
(use -paraver -t if application is multi-threaded)
```

```
% paraver app.par
```

Converting TAU Trace Files using tau2vtf to generate binary VTF3 traces with

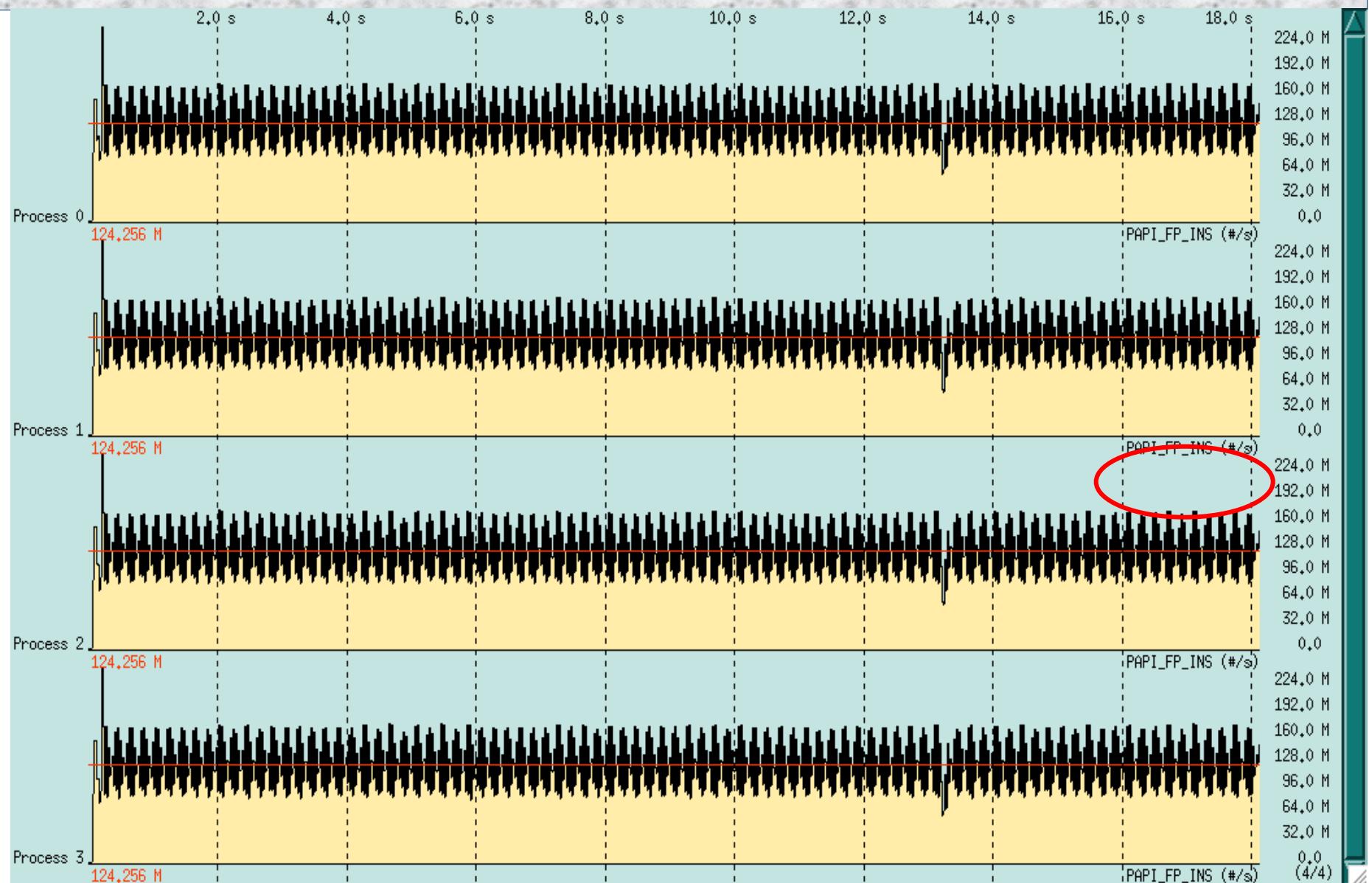
Hardware performance counter/samples data

NOTE: must configure TAU with `-vtf=dir` option in TAU v2.13.7+

```
% tau2vtf app.trc tau.edf app.vpt.gz  
% vampir app.vpt.gz
```

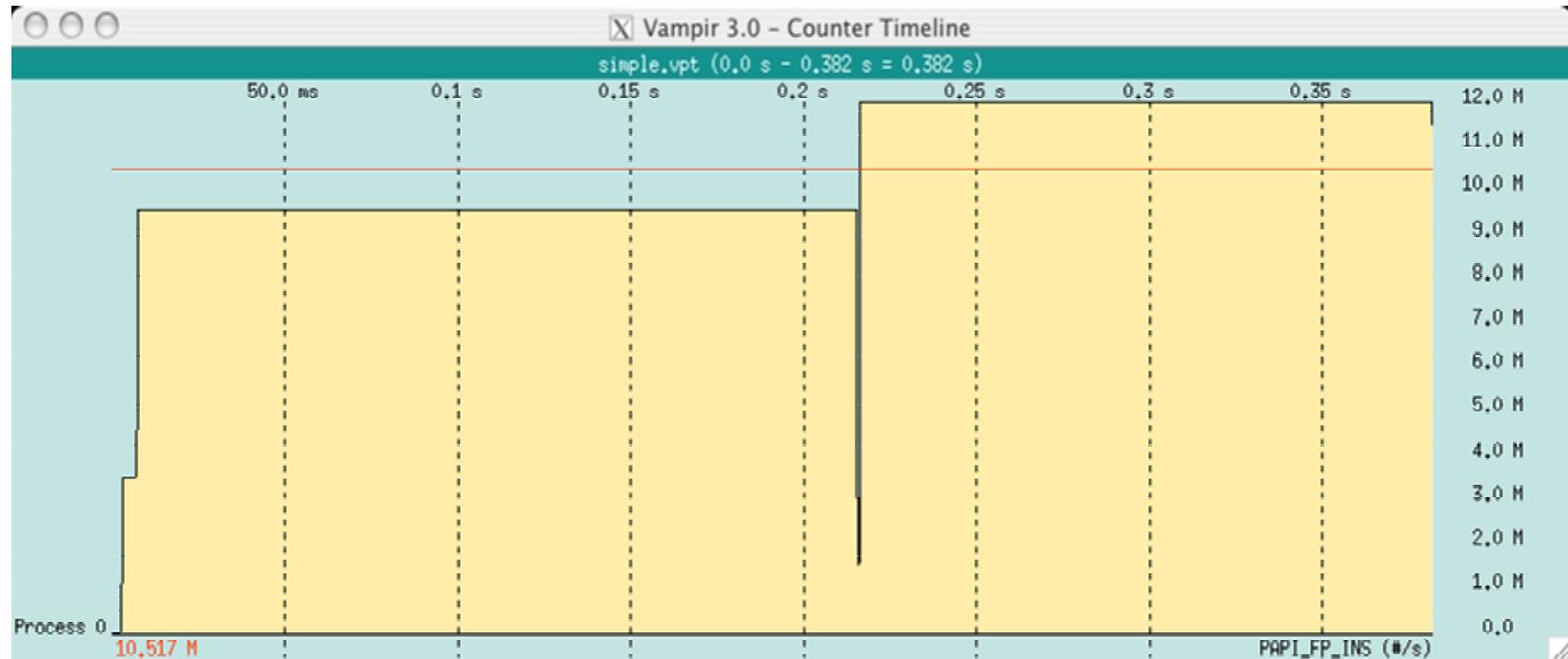


Visualizing TAU Traces with Counters/Samples





Visualizing TAU Traces with Counters/Samples





Environment Variables for Generating Traces

- With tau2vtf, TAU can automatically merge/convert traces environment variables:
 - **TAU_TRACEFILE** (name of the final VTF3 tracefile)
 - Default: not set.
% **setenv TAU_TRACEFILE app.vpt.gz**
 - **TRACEDIR** (directory where traces are stored)
 - Default: ./ or current working directory
% **setenv TRACEDIR \$SCRATCH/data/exp1**
 - **TAU_KEEP_TRACEFILES**
 - Default: not set. TAU deletes intermediate trace files
% **setenv TAU_KEEP_TRACEFILES 1**



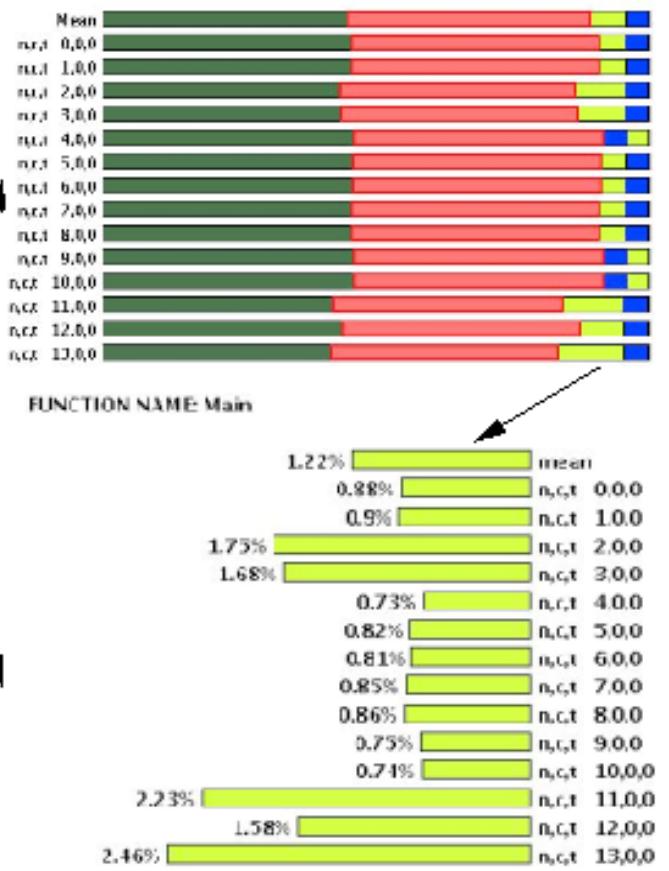
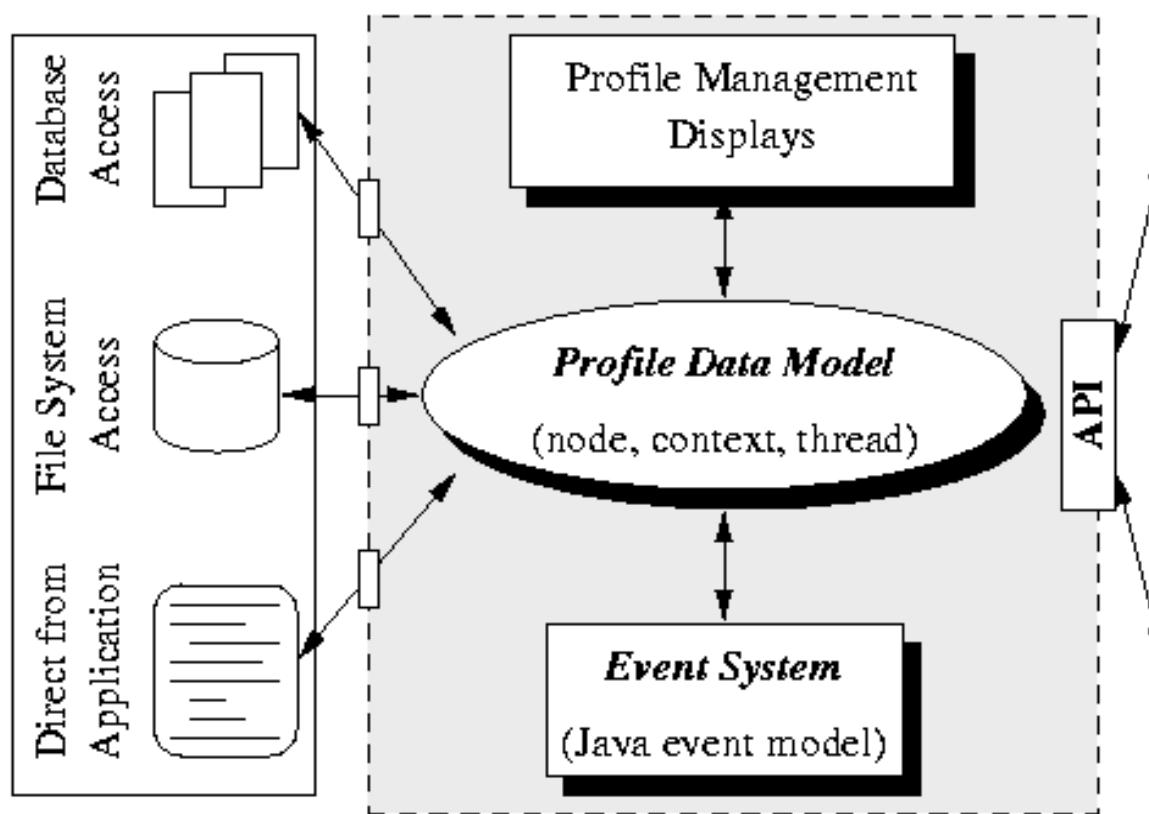
Using TAU's Environment Variables

```
% llsubmit job.sh
LoadLeveler script
/usr/bin/csh
#
#...

setenv TAU_TRACEFILE          app.vpt.gz
setenv TRACEDIR              $SCRATCH/data
setenv COUNTER1               GET_TIME_OF_DAY
setenv COUNTER2               PAPI_FP_INS
setenv COUNTER3               PAPI_TOT_CYC
...
./sp.W.4
```

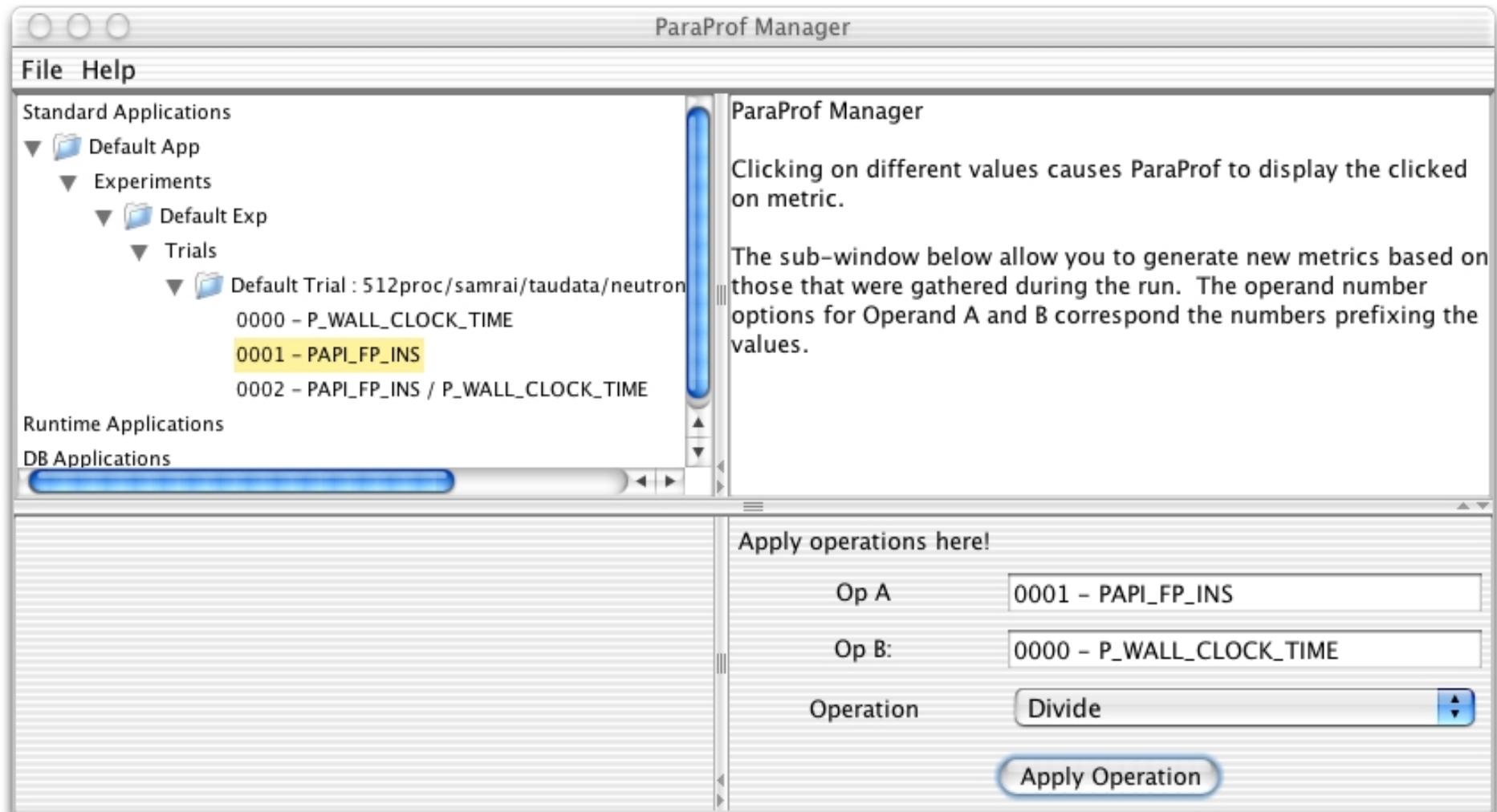
ParaProf Framework Architecture

- Portable, extensible, and scalable tool for profile analysis
- Try to offer “best of breed” capabilities to analysts
- Build as profile analysis framework for extensibility





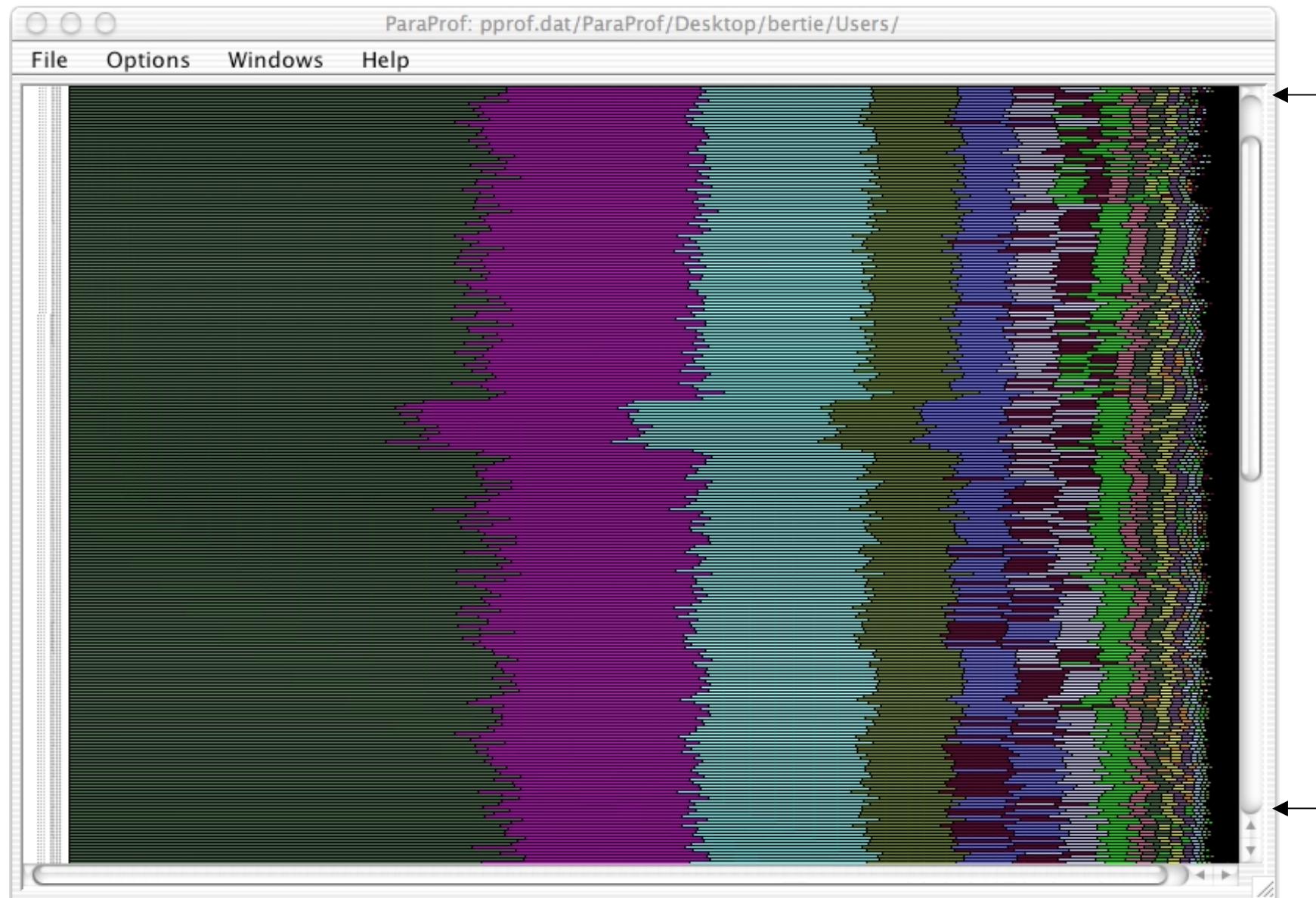
Profile Manager Window



- Structured AMR toolkit (SAMRAI++), LLNL



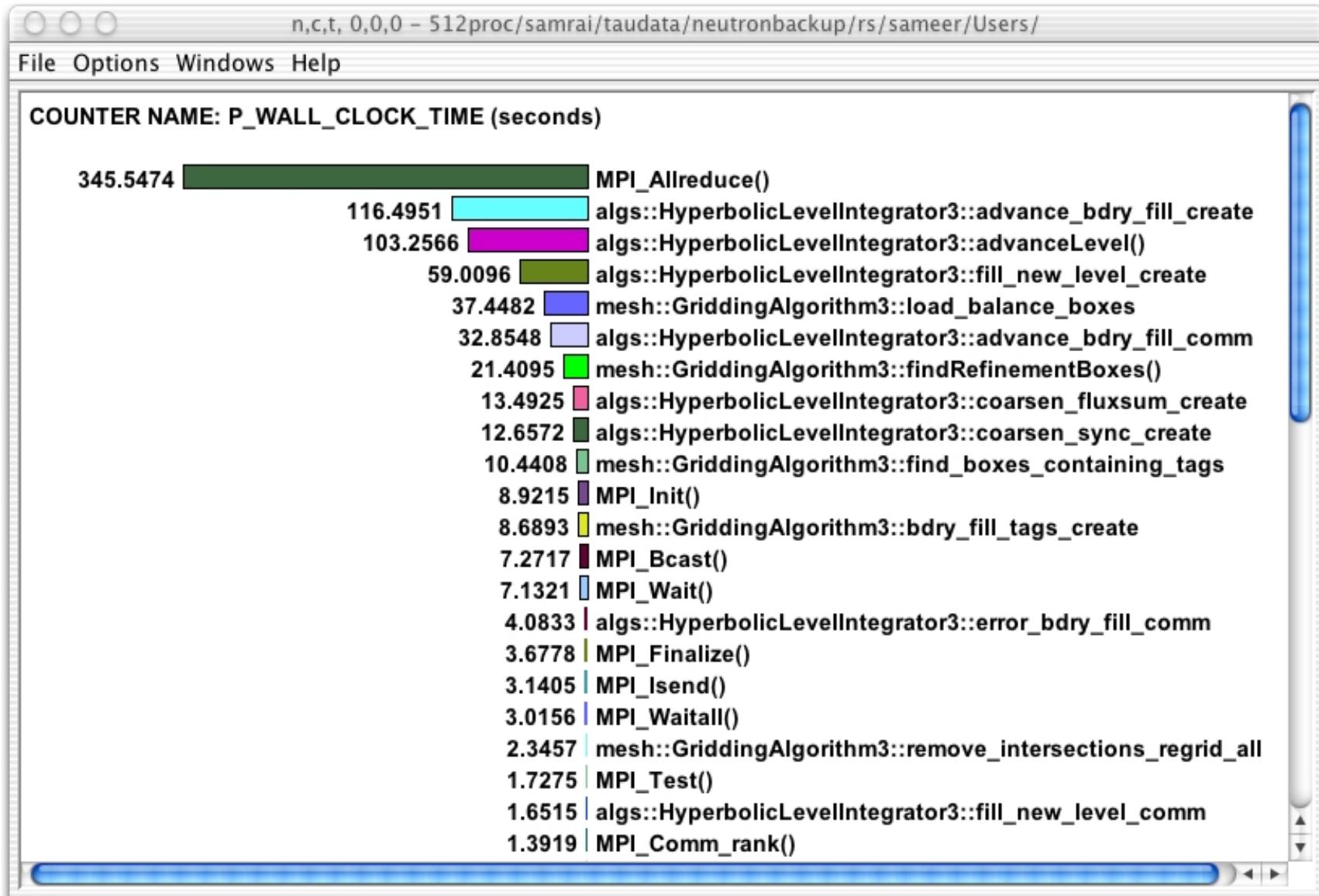
Full Profile Window (Exclusive Time)



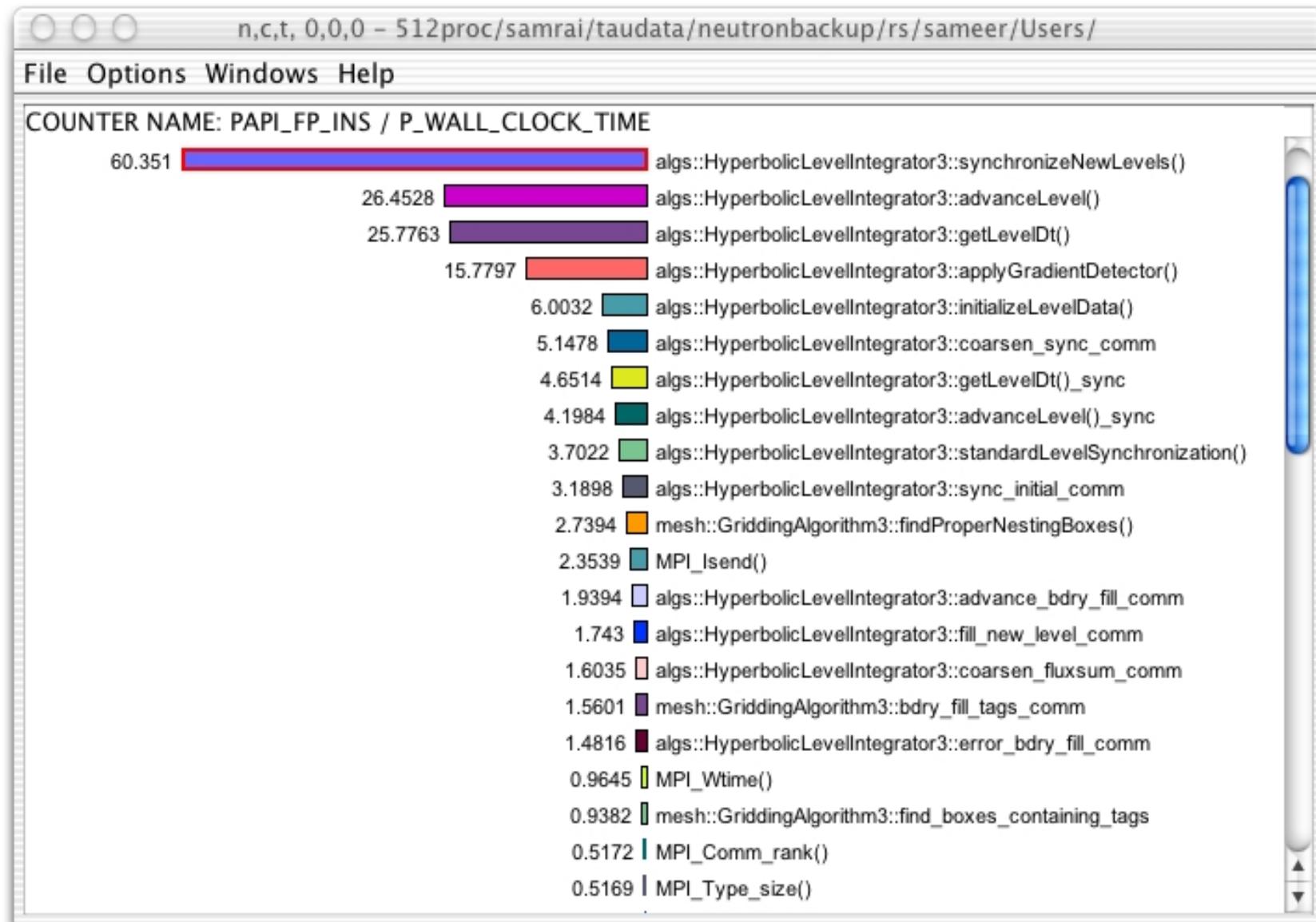
512 processes



Node / Context / Thread Profile Window

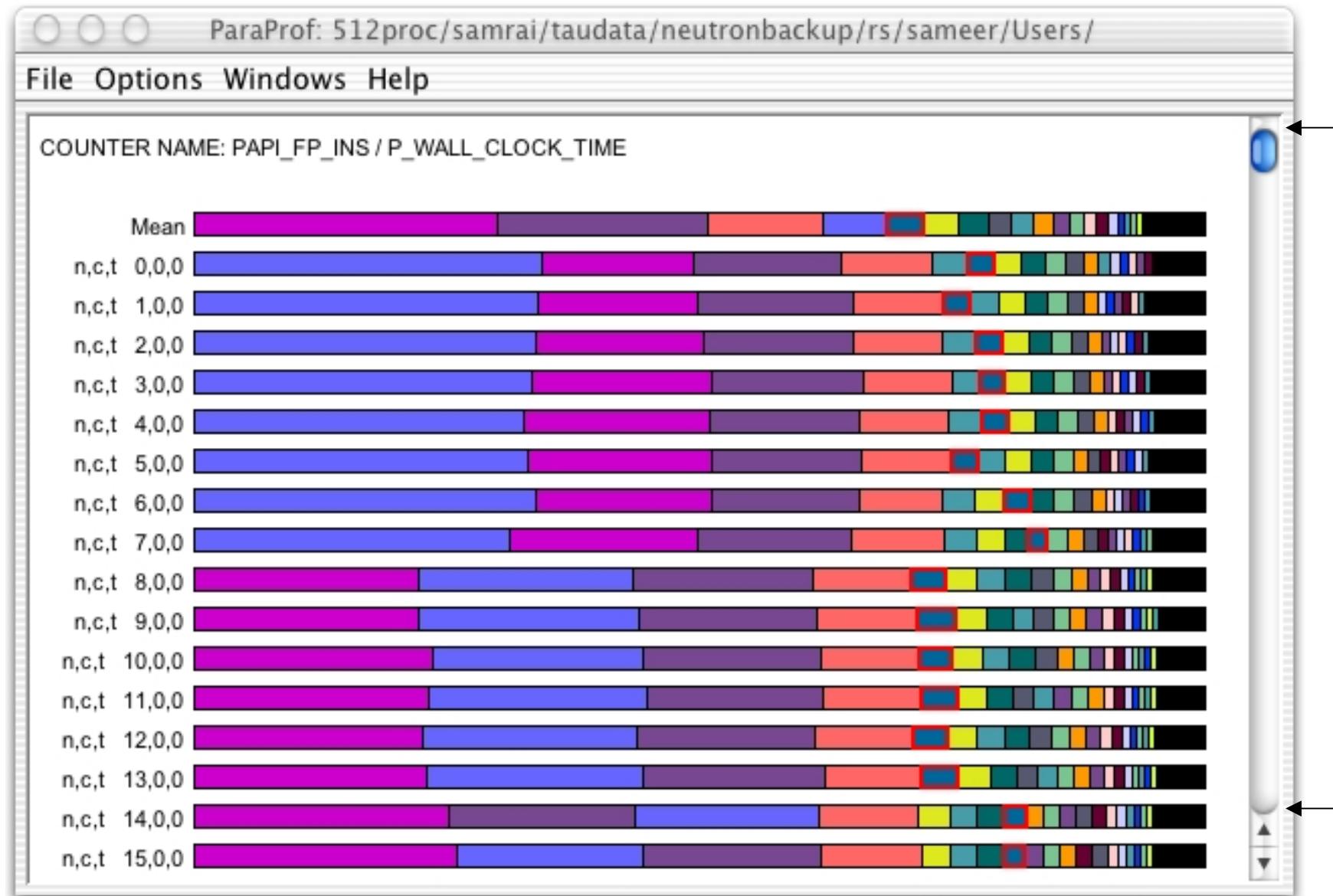


Derived Metrics



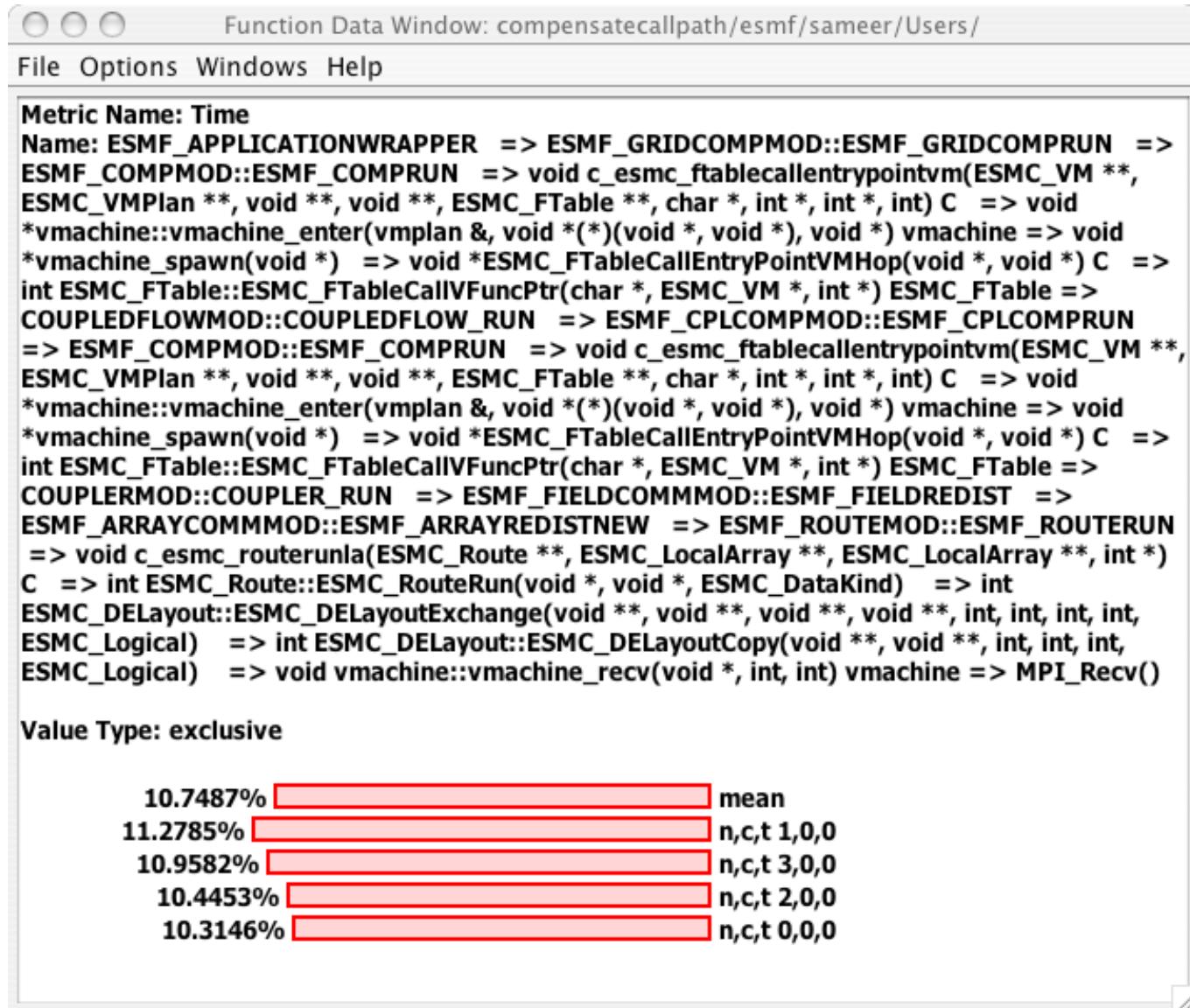


Full Profile Window (Metric-specific)



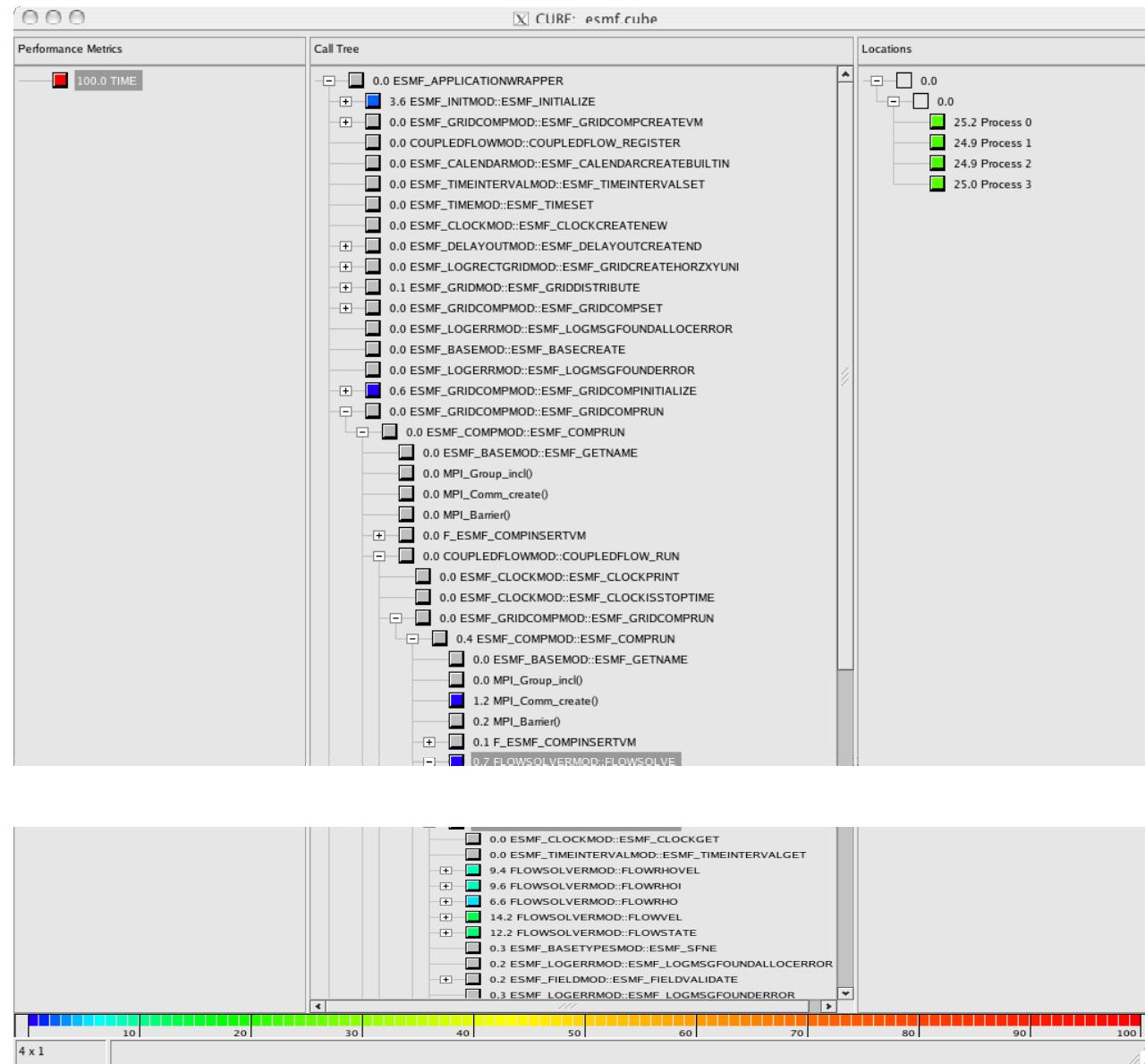


Browsing Individual Callpaths in Paraprof

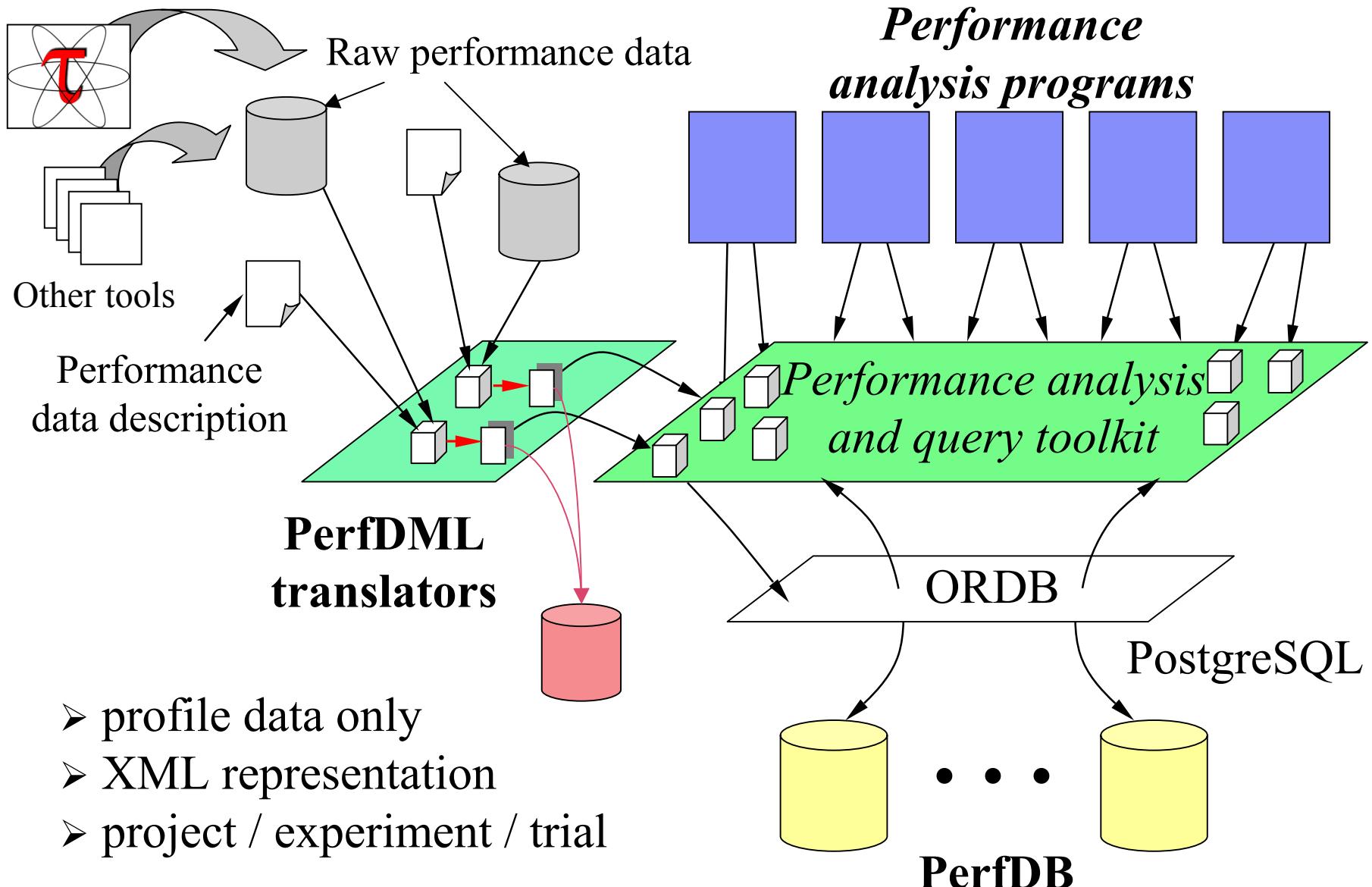




CUBE (UTK, FZJ) Browser [Sept. 2004]



TAU Performance Database Framework





TAU Performance System Status

- Computing platforms (selected)
 - IBM SP / pSeries, SGI Origin 2K/3K, Cray T3E / SV-1 / X1, HP (Compaq) SC (Tru64), Sun, Hitachi SR8000, NEC SX-5/6, Linux clusters (IA-32/64, Alpha, PPC, PA-RISC, Power, Opteron), Apple (G4/5, OS X), Windows
- Programming languages
 - C, C++, Fortran 77/90/95, HPF, Java, OpenMP, Python
- Thread libraries
 - pthreads, SGI sproc, Java, Windows, OpenMP
- Compilers (selected)
 - Intel KAI (KCC, KAP/Pro), PGI, GNU, Fujitsu, Sun, Microsoft, SGI, Cray, IBM (xlc, xlf), Compaq, NEC, Intel



Concluding Remarks

- Complex parallel systems and software pose challenging performance analysis problems that require robust methodologies and tools
- To build more sophisticated performance tools, existing proven performance technology must be utilized
- Performance tools must be integrated with software and systems models and technology
 - Performance engineered software
 - Function consistently and coherently in software and system environments
- TAU performance system offers robust performance technology that can be broadly integrated



Hands-On Session

On Seaborg

```
% module load GNU tau java vampir  
% cp -r /scratch/scratchdirs/sameer/tau ~/  
% cd ~/tau  
% tar zxf training.tar.gz  
% cd training; make  
See README, documentation and examples
```

Support Acknowledgements



- Department of Energy (DOE)
 - Office of Science contracts
 - University of Utah DOE ASCI Level 1 sub-contract
 - DOE ASCI Level 3 (LANL, LLNL)
- NSF National Young Investigator (NYI) award
- Research Centre Juelich
 - John von Neumann Institute for Computing
 - Dr. Bernd Mohr
- Los Alamos National Laboratory

